

Operating Systems and Languages Library

MS GW-BASIC Interpreter under MS-DOS

User Guide

**OLIVETTI
PERSONAL
COMPUTER**



olivetti

PREFACE

This manual is designed both as a user guide and as a reference manual for the MS GW-BASIC Interpreted language (Microsoft Rel-2.01 and 3.11) available on the Olivetti Personal Computer.

SUMMARY

This manual is divided into two parts.

Part I, the first 7 chapters, cover: start-up, modes of operation, screen editing, general programming, disk I/O, graphics, machine language subroutines, event trapping, child processes and asynchronous communications.

Part II, Chapter 8, contains a detailed description of all commands, statements, and functions available, with examples for use.

RELATED PUBLICATIONS

Installation and Operations Guide (Code 3986490 W)

MS-DOS User Guide (Code 4001410 G)

MS GW-BASIC Compiler User Guide (Code 4021580 F)

DISTRIBUTION: General (G)

FIFTH EDITION: February 1986

*Copyright © Microsoft Corporation
1980 ÷ 1985*

*Copyright © 1986 by Olivetti
All rights reserved*

PUBLICATION ISSUED BY:

Ing. C. Olivetti & C., S.p.A.
Direzione Documentazione
77, Via Jervis - 10015 Ivrea — Italy

USING YOUR SYSTEM AS A CALCULATOR	2-19
ENTERING A PROGRAM	2-20
AUTOMATIC LINE NUMBERING	2-21
LISTING A PROGRAM	2-22
SAVING A PROGRAM	2-22
LOADING A PROGRAM	2-24
EXECUTING A PROGRAM	2-24
RUNNING A SAMPLE PROGRAM	2-25
PROGRAM INTERRUPTS	2-27
 3. CONSTANTS, VARIABLES AND EXPRESSIONS	
CONSTANTS	3-1
SINGLE AND DOUBLE PRECISION FORM FOR NUMERIC CONSTANTS	3-2
VARIABLES	3-3
VARIABLE NAMES AND DECLARATION CHARACTERS	3-3
ARRAY VARIABLES	3-5
MEMORY REQUIREMENTS	3-5
TYPE CONVERSION	3-6
EXPRESSIONS AND OPERATORS	3-8
ARITHMETIC OPERATORS	3-8
INTEGER DIVISION AND MODULUS ARITHMETIC	3-10
OVERFLOW	3-10

RELATIONAL OPERATORS	3-11
LOGICAL OPERATORS	3-12
FUNCTIONAL OPERATORS	3-17
STRING OPERATORS	3-17
4. DISK FILE HANDLING	
INTRODUCTION	4-1
DEVICE INDEPENDENT INPUT/OUTPUT	4-1
HOW MS-DOS KEEPS TRACK OF YOUR FILES	4-1
FILE NUMBERS	4-2
NAMING FILES	4-2
NAMING DEVICES	4-3
DIRECTORY PATHS	4-4
CURRENT DIRECTORY	4-7
COMMANDS FOR PROGRAM FILES	4-7
PROTECTED FILES	4-9
DISK DATA FILES - SEQUENTIAL AND RANDOM ACCESS	4-9
SEQUENTIAL FILES	4-10
CREATING A SEQUENTIAL FILE	4-10
ACCESSING A SEQUENTIAL FILE	4-12
ADDING DATA TO A SEQUENTIAL FILE	4-13
RANDOM ACCESS FILES	4-13
CREATING A RANDOM ACCESS FILE	4-14

ACCESSING A RANDOM ACCESS FILE	4-15
FILE LOCKING	4-19
5. GRAPHICS	
SELECTING THE GRAPHICS ENVIRONMENT	5-1
TEXT MODE	5-2
GRAPHICS MODES	5-4
MEDIUM RESOLUTION MODE	5-5
HIGH RESOLUTION MODE	5-6
SUPER RESOLUTION MODE	5-7
WINDOWS AND VIEWPORTS	5-8
TRANSFORMATION USING WINDOW AND VIEW	5-9
COORDINATES	5-10
ANIMATION TECHNIQUES	5-11
6. ADVANCED FEATURES	
MACHINE LANGUAGE SUBROUTINES	6-1
MEMORY ALLOCATION	6-1
LOADING SUBROUTINES INTO MEMORY	6-2
USING THE POKE STATEMENT	6-2
USING THE BLOAD COMMAND	6-3
CALLING THE SUBROUTINE FROM GW-BASIC	6-4
CALL STATEMENT	6-4

CALLS STATEMENT	6-9
USR FUNCTION	6-9
EVENT TRAPPING	6-12
THE ON GOSUB STATEMENT	6-14
THE RETURN STATEMENT	6-14
GW-BASIC AND CHILD PROCESSES	6-15
HARDWARE	6-16
THE FILE SYSTEM	6-16
MEMORY MANAGEMENT	6-17
 7. ASYNCHRONOUS COMMUNICATIONS	
OPENING COMMUNICATIONS FILES	7-1
COMMUNICATION I/O	7-1
COMMUNICATION I/O FUNCTIONS	7-1
THE INPUT\$ FUNCTION FOR COMMUNICATION FILES	7-2
AN EXERCISE IN COMMUNICATION I/O	7-3
 PART II	
 8. COMMANDS, STATEMENTS AND FUNCTIONS	
INTRODUCTION	8-1
COMMANDS	8-2
STATEMENTS	8-5
NON-I/O STATEMENTS	8-5

I/O STATEMENTS	8-11
FUNCTIONS	8-18
NUMERIC FUNCTIONS (returning a numeric value)	8-18
STRING FUNCTIONS (returning a string value)	8-23
ABS Function	8-26
ASC Function	8-27
ATN Function	8-28
AUTO Command	8-29
BEEP Statement	8-30
BLOAD Command	8-31
BSAVE Command	8-33
CALL Statement	8-35
CALLS Statement	8-36
CDBL Function	8-37
CHAIN Statement	8-38
CHDIR Command	8-43
CHR\$ Function	8-45
CINT Function	8-46
CIRCLE Statement	8-47
CLEAR Command	8-53
CLOSE Statement	8-56

CLS Statement	8-57
COLOR Statement (Text Mode)	8-59
COLOR Statement (Medium-resolution Graphics)	8-62
COLOR Statement (High-resolution Graphics)	8-66
COLOR Statement (Super-resolution Graphics)	8-69
COM(<i>n</i>) Statement	8-71
COMMON Statement	8-73
CONT Command	8-77
COS Function	8-78
CSNG Function	8-79
CSRLIN Function	8-80
CVI, CVS, CVD Functions	8-80
DATA Statement	8-82
DATE\$ Function and Statement	8-84
DEF FN Statement	8-86
DEF SEG Statement	8-88
DEF USR Statement	8-90
DEFINT / SNG / DBL / STR Statements	8-92
DELETE Command	8-93
DIM Statement	8-94
DRAW Statement	8-99
EDIT Command	8-105
END Statement	8-106

ENVIRON Statement	8-107
ENVIRON\$ Function	8-109
EOF Function	8-111
ERASE Statement	8-112
ERDEV and ERDEV\$ Functions	8-113
ERR and ERL Functions	8-115
ERROR Statement	8-117
EXP Function	8-119
FIELD Statement	8-120
FILES Command	8-123
FIX Function	8-125
FOR...NEXT Statements	8-126
FRE Function	8-130
GET (COM files) Statement	8-131
GET (Files) Statement	8-132
GET (Graphics) Statement	8-133
GOSUB...RETURN Statements	8-136
GOTO Statement	8-138
GW BASIC Command	8-139
HEX\$ Function	8-146
IF...GOTO...ELSE and IF...THEN... ELSE Statements	8-147
INKEY\$ Function	8-150
INP Function	8-152

INPUT Statement	8-153
INPUT # Statement	8-156
INPUT\$ Function	8-157
INSTR Function	8-160
INT Function	8-162
IOCTL Statement	8-162
IOCTL\$ Function	8-165
KEY Statement	8-166
KEY(<i>n</i>) Statement	8-171
KILL Command	8-174
LCOPY Command	8-175
LEFT\$ Function	8-176
LEN Function	8-177
LET Statement	8-178
LINE Statement	8-179
LINE INPUT Statement	8-182
LINE INPUT # Statement	8-184
LIST Command	8-185
LLIST Command	8-187
LOAD Command	8-188
LOC Function	8-190
LOCATE (Text) Statement	8-191
LOCATE (Graphics) Statement	8-195

LOCK Statement	8-199
LOF Function	8-201
LOG Function	8-202
LPOS Function	8-203
LPRINT and LPRINT USING Statement	8-204
LSET and RSET Statements	8-205
MERGE Command	8-207
MID\$ Function and Statement	8-208
MKDIR Command	8-211
MKI\$, MKS\$, MKD\$ Functions	8-213
NAME Command	8-215
NEW Command	8-217
OCT\$ Function	8-217
ON COM(<i>n</i>) GOSUB Statement	8-218
ON ERROR GOTO Statement	8-221
ON...GOSUB and ON...GOTO Statements	8-224
ON KEY(<i>n</i>) GOSUB Statement	8-225
ON PLAY(<i>n</i>) GOSUB Statement	8-229
ON TIMER (<i>n</i>) GOSUB Statement	8-232
OPEN Statement	8-234
OPEN COM Statement	8-242
OPTION BASE Statement	8-246
OUT Statement	8-247
PAINT Statement	8-248

PEEK Function	8-253
PLAY Statement	8-254
PLAY(<i>n</i>) Function	8-258
PLAY {ON OFF STOP} Statement	8-259
PMAP Function	8-260
POINT Function	8-263
POKE Statement	8-266
POS Function	8-267
PRESET Statement	8-268
PRINT Statement	8-269
PRINT USING Statement	8-272
PRINT # and PRINT # USING Statements	8-279
PSET Statement	8-281
PUT (COM files) Statement	8-283
PUT (Files) Statement	8-284
PUT (Graphics) Statement	8-286
RANDOMIZE Statement	8-290
READ Statement	8-291
REM Statement	8-294
RENUM Command	8-296
RESET Command	8-297
RESTORE Statement	8-298
RESUME Statement	8-299

PART I

1. PROGRAMMING IN GW-BASIC - GENERAL

ABOUT THIS CHAPTER

This chapter introduces you to GW-BASIC, highlighting its major features, system requirements, the line format, the character set, and reserved words.

CONTENTS

INTRODUCTION	1-1
GW-BASIC MAJOR FEATURES	1-1
SYSTEM REQUIREMENTS	1-2
SYNTAX CONVENTIONS	1-2
LINE FORMAT	1-5
LINE NUMBERS	1-6
CHARACTER SET	1-6
RESERVED WORDS	1-8

INTRODUCTION

The GW-BASIC language is the most extensive implementation of BASIC available for personal computers. It meets the requirements of the ANSI standard for BASIC, and supports many unique features rarely found in other BASICs. In addition, GW-BASIC provides sophisticated string handling and structured programming features that are especially suited for applications development. The GW-BASIC language has improved graphics possibilities, and gives users what they want from a BASIC - ease of use plus the features that make a personal computer perform like a minicomputer.

GW-BASIC MAJOR FEATURES

GW-BASIC is a high level language which provides easy ways of solving commercial and scientific problems.

Some of the main features of GW-BASIC are as follows:

- Re-direction of Standard Input (INPUT, LINE INPUT, INPUT\$, IN-KEY\$) and Standard Output (PRINT, PRINT USING)
- Tree-structured directories for organization of archives of programs and data and directory management support commands (MKDIR/CHDIR/RMDIR)
- Use of a powerful Screen Editor
- High-level Graphics: three resolution modes (Medium, High and Super)
- Calls to Assembly language routines, using the CALL or CALLS instruction or the USR function
- Error management and event trapping using user-defined routines
- RS-232-C standard communication interface management
- Single or double precision mathematical functions (ATN, COS, LOG, EXP, SIN, SQR, TAN)
- Control of GW-BASIC's memory allocation for user routines with the /M: switch in the GWBASIC command

- Chaining with common variables: chaining is used to allow programs larger than the available memory
- Declaration statements: variables names can be listed in a declaration statement explicitly specifying the variable to be an integer variable, a single or double precision variable, or a string variable.

SYSTEM REQUIREMENTS

GW-BASIC, under the MS-DOS operating system, can be run using the minimum system configuration.

A minimum of one disk drive is required.

SYNTAX CONVENTIONS

1. Uppercase letters and words, and the symbols listed below, should be typed in the actual line exactly as shown.

(
)
,
;
:
,,
=
/
\

\$
.
<
>

In the statement:

WRITE # *filenum*, *list-of-expressions*

WRITE , # , and the comma (,) after *filenum* should be typed as shown.

PROGRAMMING IN GW-BASIC - GENERAL

2. Lowercase letters and words represent "variable information" (or "parameters") that the user must provide.

In the statement:

KILL *filespec*

filespec should be replaced by a specific value; for example, "myfile".

3. The symbols listed below are used to define the syntax of a line, but should not be typed in the actual line:

- | vertical stroke ("or" sign), to indicate alternatives
- { } braces, to indicate a choice
- [] brackets, to indicate optional
- ... ellipsis, to indicate repetition
- hyphen, to join multiple-name parameters.

In some statements or commands (e.g. LIST, LLIST etc...) the hyphen is used as an operator to separate parameters. In this case bold face is used to distinguish hyphens that are used for this purpose from hyphens used to join multiple-name parameters.

4. Braces group related items (divided by a vertical stroke), such as alternatives.

The representation:

{ **A**|**B**|**C** }

indicates that you must choose one of the items enclosed within the braces.

The representation:

A
or
B
or
C

can also be used.

5. Brackets also group related items (divided by a vertical stroke); however, everything within the brackets is optional and may be omitted.

The representation:

[A|B|C]

indicates that you may choose one of the items enclosed within the brackets or that you may omit all of the items.

6. An ellipsis indicates that the preceding item or group of items may be repeated more than once in succession.

The representation:

A [,B] ...

indicates that A can be typed alone or can be followed by

,B

more than once in succession.

The representation:

A [,*list-of-B*]

is also permitted and has the same meaning as

A [,B] ...

7. Characters which appear in a listing in bold face represent characters entered through the keyboard. GW-BASIC lines you are entering from the keyboard are also represented in bold face.

PROGRAMMING IN GW-BASIC - GENERAL

LINE FORMAT

GW-BASIC lines may contain a maximum of 254 characters and have the following format:

`[nnnnn] statement [:statement] ... ['comment] CR`

A GW-BASIC program line always begins with a line number (an unsigned integer in the range 1 to 65,529), and ends with a carriage return (`CR`). A program line is stored in memory as soon as you enter `CR`.

A GW-BASIC immediate line, i.e. a line that is executed as soon as you enter it, always begins with a letter, as you have to omit the line number in this case.

More than one GW-BASIC statement may be placed on a line, but each successive statement must be separated from the last by a colon.

At the end of a GW-BASIC line (before `CR`) you may enter a comment string preceded by a single quotation mark (`'`).

A comment string preceded either by the keyword `REM` or by a single quotation mark may also be written just after the line number.

It is possible to extend a logical line over more than one physical line by using the line feed (`CTRL CR`). Line feed lets you continue typing a logical line on the next physical line without entering a `CR` . From now on, in the interest of brevity, we shall not specify `CR` at the end of a GW-BASIC line.

Examples

```
10 FOR K = 1 TO 20
```

is a GW-BASIC program line

```
100 GOSUB 1000 'branch to SUB1
```

is a GW-BASIC program line with a comment at the end

```
1000 'SUB1
```

is a GW-BASIC program line which contains only a comment

PRINT A\$

is a GW-BASIC immediate line.

LINE NUMBERS

Every GW-BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory. Line numbers are also used as references in branching and editing. Line numbers must be in the range 0 to 65,529.

For the EDIT, LIST, AUTO, and DELETE commands, a period (.) may be used to reference the current line.

CHARACTER SET

GW-BASIC recognizes the following sets of characters:

- Alphabetic characters (upper and lower case letters of the alphabet)
- Numeric characters (the digits 0 through 9)
- Special characters (see the following page)

PROGRAMMING IN GW-BASIC - GENERAL

Special Characters

The list of GW-BASIC special characters is summarized below:

CHARACTER	ACTION
	Blank
=	Equals sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(Left parenthesis
)	Right parenthesis
%	Percent sign or integer type declaration character
#	Number (or pound) sign or double precision type declaration character
\$	Dollar sign or string type declaration character
!	Exclamation point or single precision type declaration character
[Left bracket
]	Right bracket
,	Comma
.	Period or decimal point
'	Single quotation mark (apostrophe)
"	Double quotation mark (string delimiter)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark (PRINT abbreviation)
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At sign
_	Underscore

RESERVED WORDS

GW-BASIC comprises a set of statements, commands, function names, and operator names which are treated as reserved words, and which cannot be used as variable names. The total list of GW-BASIC reserved words is as follows:

ABS	DELETE	IOCTL\$
AND	DIM	KEY
ASC	DRAW	KILL
ATN	EDIT	LEFT\$
AUTO	ELSE	LEN
BEEP	END	LET
BLOAD	ENVIRON	LINE
BSAVE	ENVIRON\$	LIST
CALL	EOF	LLIST
CALLS	EQV	LOAD
CDBL	ERASE	LOC
CHAIN	ERDEV	LOCATE
CHDIR	ERDEV\$	LOCK
CHR\$	ERL	LOF
CINT	ERR	LOG
CIRCLE	ERROR	LPOS
CLEAR	EXP	LPRINT
CLOSE	FIELD	LSET
CLS	FILES	MERGE
COLOR	FIX	MID\$
COM	FNxxxxxxxx	MKDIR
COMMON	FOR	MKD\$
CONT	FRE	MKI\$
COS	GET	MKS\$
CSNG	GOSUB	MOD
CSRLIN	GOTO	NAME
CVD	HEX\$	NEW
CVI	IF	NEXT
CVS	INKEY\$	NOT
DATA	INP	OCT\$
DATE\$	INPUT	OFF
DEF	INPUT\$	ON
DEFDBL	INPUT #	OPEN
DEFINT	INSTR	OPTION
DEFSGN	INT	OR
DEFSTR	IOCTL	OUT

PROGRAMMING IN GW-BASIC - GENERAL

PAINT	TIMES\$
PEEK	TO
PLAY	TROFF
PMAP	TRON
POINT	UNLOCK
POKE	USING
POS	USR
PRESET	VAL
PRINT	VARPTR
PRINT #	VARPTR\$
PSET	VIEW
PUT	WAIT
RANDOMIZE	WEND
READ	WHILE
REM	WIDTH
RENUM	WINDOW
RESET	WRITE
RESTORE	WRITE #
RESUME	XOR
RETURN	
RIGHT\$	
RMDIR	
RND	
RSET	
RUN	
SAVE	
SCREEN	
SGN	
SHELL	
SIN	
SOUND	
SPACE\$	
SPC	
SQR	
STEP	
STOP	
STR\$	
STRING\$	
SWAP	
SYSTEM	
TAB	
TAN	
THEN	
TIMER	

2. GETTING STARTED

ABOUT THIS CHAPTER

This chapter explains how to enter and leave GW-BASIC, the modes of operation, the use of the keyboard and the GW-BASIC Screen Editor. It also explains how to enter, list, save, load, modify, and execute a GW-BASIC program.

CONTENTS

INITIALIZATION PROCEDURE	2-1	USING YOUR SYSTEM AS A CALCULATOR	2-19
LEAVING GW-BASIC	2-2	ENTERING A PROGRAM	2-20
MODES OF OPERATION	2-2	AUTOMATIC LINE NUMBERING	2-21
KEYBOARD	2-3	LISTING A PROGRAM	2-22
FUNCTION KEYS	2-3	SAVING A PROGRAM	2-22
TYPEWRITER KEYBOARD	2-4	LOADING A PROGRAM	2-24
NUMERIC KEYPAD	2-7	EXECUTING A PROGRAM	2-24
THE GW-BASIC SCREEN EDITOR	2-8	RUNNING A SAMPLE PROGRAM	2-25
SPECIAL SCREEN EDITOR KEYS	2-8	PROGRAM INTERRUPTS	2-27
CORRECTING THE CURRENT LINE	2-13		
MODIFYING PROGRAM LINES	2-17		

GETTING STARTED

INITIALIZATION PROCEDURE

To start GW-BASIC, the MS-DOS operating system must first be installed. When MS-DOS has been installed and the system prompt:

A>

is displayed, enter the GWBASIC command:

GWBASIC

to load GW-BASIC from the diskette inserted in drive A into memory.

Upon loading, GW-BASIC you can:

- insert a diskette containing your GW-BASIC programs and execute a program. For example you can enter:

run "b:myfile"

if "myfile" is the GW-BASIC program you want to run from the diskette inserted in drive B

- enter GW-BASIC program or immediate lines (see "Modes of Operation" below). For example:

10 K = 10

20 FOR J = 1 TO 2

etc...

These are program lines. When you enter them, they are stored in memory to form a program. To execute it, you must enter the RUN command.

Remarks

The GWBASIC command may be entered with several options to optimize memory occupation, redirecting standard input or output, etc. (See the GWBASIC command in Chapter 8 for details.)

LEAVING GW-BASIC

To exit from GW-BASIC and return to MS-DOS, enter:

SYSTEM

This closes all data files before returning to MS-DOS; your GW-BASIC program is lost, while MS-DOS remains resident.

When the MS-DOS system prompt (`A>`) appears once more, you can enter another MS-DOS command.

MODES OF OPERATION

The GW-BASIC Interpreter may be used in either of two modes: "immediate mode" or "program mode".

In "immediate" (or "direct") mode, statements and commands are executed as they are entered. They are not preceded by line numbers. After each direct statement followed by a carriage return, the screen will display the "Ok" prompt. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. Immediate mode is useful for debugging and for using the GW-BASIC Interpreter as a calculator for quick computations that do not require a complete program.

Example

```
Ok
PRINT 45 + 3
48
Ok
```

Program (or "indirect") mode is used for entering programs. Program lines are preceded by line numbers and are stored in memory as soon as they are entered. The program stored in memory is executed by entering the RUN command.

PROGRAMMING IN GW-BASIC - GENERAL

Example

```
Ok
10 PRINT 45 + 3
RUN
48
Ok
```

KEYBOARD

Two different types of keyboard are available for the Olivetti PC: KEYBOARD 1 and KEYBOARD 2. All the keyboards have several national versions. See the "Installation and Operation" manual for a full description of the keyboard(s) provided with your Olivetti PC.

FUNCTION KEYS

There are 10 function keys on Keyboard 1 and 18 on Keyboard 2.

These function keys can be tailored to the User's needs using the KEY and ON KEY(n) GOSUB statements.

The KEY statement can be used to assign a specific command or sequence of characters to a function key, other than the pre-assigned standard commands. The ON KEY(n) GOSUB statement can be used to generate program interrupts via a specified function key.

Refer to Chapter 8 for further details.

TYPEWRITER KEYBOARD

The standard typewriter keyboard, located in the center, is used to enter letters, numbers, special characters, and control characters.

Shift Keys

If you want to enter upper case letters or the upper symbol on those keys containing two symbols, hold down one of the two ↑ keys (on Keyboard 1), or one of the two **SHIFT** keys (on Keyboard 2), and press the corresponding key.

From now on we shall always refer to the ↑ keys as **SHIFT** keys by convention.

Carriage Return Key

The carriage return key is identified by the symbol ↵. By convention we shall refer to this key as the **CR** key.

You must press **CR** to close a GW-BASIC line and send it to the system for processing.

Shift Lock for Letters

You can enable or disable Shift Lock for letters (A-Z) by pressing **CAPS LOCK**.

The **CAPS LOCK** key is similar to a typewriter Shift Lock Key, but it only gives you uppercase letters, and will not give you the upper symbols on the numeric or other keys.

GETTING STARTED

Backspace

The backspace key ← (on Keyboard 1) or **BS** (on Keyboard 2) moves the cursor one position to the left erasing the last character you have typed.

To move the cursor to the left without erasing any characters, you should use the Cursor Left Key.

Control Characters

You can generate control characters by holding down the **CTRL** or **ALT** key while pressing another key. GW-BASIC recognizes the following control characters:

CONTROL CHARACTER	ACTION
CTRL BREAK	CTRL BREAK can be used for different purposes: 1. To interrupt the program at the following GW-BASIC instruction and return to GW-BASIC Command Level. 2. To cancel automatic line numbering mode while entering a program. 3. To return to Command Level, without saving any changes that you made to the current line.
CTRL G	Sounds the bell.
CTRL NUM LOCK (on Keyboard 1) CTRL FUNCT LOCK (on Keyboard 2)	Causes the system to 'pause' so as to temporarily halt printing or program listing. The pause continues until you press any key (except SHIFT , CTRL or ALT).

CONTROL CHARACTER	ACTION
CTRL T	Scrolls the function key display horizontally across the screen (on the 25th screen line), when the width is 40. When the width is 80, it toggles the function key display ON and OFF.
ALT CTRL DEL	Performs a System Reset by holding down the CTRL and ALT keys, and then pressing the DEL key.
CTRL PR T SC (on Keyboard 1) CTRL SCR PR T (on Keyboard 2)	<p>All text sent to the screen is also sent to the system printer. You can stop printing by repeating the key sequence.</p> <p>If you press PRTSC while holding down SHIFT on Keyboard 1, or if you press SCR PRT on Keyboard 2, MS-DOS will make a single printed copy of the entire display screen.</p> <p>To print both text and graphics, you have to use the MS-DOS GRAPHICS command before entering GW-BASIC.</p>
CTRL L	Outputs a formfeed character. It has the same function as the CLS statement, i.e. it clears the screen or the current graphics viewport (if a viewport has been defined).
CTRL Z	Sets an end of file condition. See the OPEN COM statement.

Other control characters are described in the subsection entitled "Special Screen Editor Keys" later in this chapter.

GETTING STARTED

Direct Entry of GW-BASIC Keywords

A GW-BASIC Keyword is entered by holding down the **ALT** key while pressing one of the alphabetic keys (A - Z). Keywords associated with each letter are listed below.

A - AUTO	N - NEXT
B - BSAVE	O - OPEN
C - COLOR	P - PRINT
D - DELETE	Q - ****
E - ELSE	R - RUN
F - FOR	S - SCREEN
G - GOTO	T - THEN
H - HEX\$	U - USING
I - INPUT	V - VAL
J - ****	W - WIDTH
K - KEY	X - XOR
L - LOCATE	Y - ****
M - MERGE	Z - ****

**** = unused keys

NUMERIC KEYPAD

A group of 15 keys on Keyboard 1 and 27 keys on Keyboard 2, at the right-hand side of the keyboard. It is arranged much like a standard calculator's keypad and is called the "numeric keypad". It includes not only the numbers 0 through 9, the decimal point, the plus (+) and minus (-) keys, but also cursor movement keys, **PGUP**, **PGDN**, **HOME**, **NUM LOCK** (on Keyboard 1), **FUNCT LOCK** (on Keyboard 2), **SCROLL LOCK** (on Keyboard 1), **SCROLL ON** (on Keyboard 2), **BREAK**, **END**, **INS**, **DEL**, etc.

Note that some keys like **SCROLL LOCK**, **PGUP**, and **PGDN** etc., are not used by GW-BASIC, but you may assign meanings to them within a GW-BASIC program.

Number Lock State (Keyboard 1)

You can press the **NUM LOCK** key (on Keyboard 1) to shift the numeric keypad into upper-case. This mode provides the numbers 0 through 9 and the decimal point. (Holding down one of the two **SHIFT** keys produces the corresponding lower-case keys in this mode.) To return to lower-case, press **NUM LOCK** once again.

Function Lock State (Keyboard 2)

You can press the **FUNCT LOCK** key (on Keyboard 2) to shift the numeric keypad into the lower-case. This mode provides the functions (like **HOME**, **INS**, **DEL**, **END**, **PGUP**, **PGDN**). Holding down one of the two **SHIFT** keys produces the corresponding upper-case keys in this mode. To return to upper-case, press **FUNCT LOCK** once again.

THE GW-BASIC SCREEN EDITOR

All text entered while GW-BASIC is at command level is processed by the GW-BASIC Editor. This is a "screen line editor" which allows you to change a line anywhere on the screen (only one line at a time). It is important to note that changes are only registered when you press **CR** on that line.

SPECIAL SCREEN EDITOR KEYS

The GW-BASIC Editor recognizes 9 Numeric Keypad keys, the Backspace key, and the **CTRL** key, to move the cursor, insert or delete characters.

The keys and their functions are listed below.

KEY	FUNCTION
HOME	HOME : Positions the cursor in the top left hand corner of the screen.
CTRL HOME	CLEAR SCREEN : Clears the screen and moves the cursor to the "Home" position.
↑	CURSOR UP : Moves the cursor up one line.

GETTING STARTED

KEY	FUNCTION
↓	CURSOR DOWN: Moves the cursor one position down.
←	CURSOR LEFT: Moves the cursor one position left. When the cursor is moved beyond the left limit of the screen, it appears at the right side of the screen on the preceding line.
→	CURSOR RIGHT: Moves the cursor one position right. If the cursor is moved beyond the right limit of the screen, it appears at the left side of the screen on the following line.
CTRL →	<p>NEXT WORD: Moves the cursor to the beginning of the following word, i.e., to the next character to the right of the cursor in the set [A..Z] or [a..z] or [0..9], which follows a blank or special character.</p> <p>For example, in the following line:</p> <pre>30 IF <u>L</u> < = 0 THEN 20</pre> <p>The cursor is under the letter L. If you press CTRL →, the cursor will move to the beginning of the next word, which is 0:</p> <pre>30 IF L < = <u>0</u> THEN 20</pre> <p>If you press CTRL → again, the cursor will move to the next word, which is THEN:</p> <pre>30 IF L < = 0 <u>T</u>HEN 20</pre>

KEY	FUNCTION
CTRL ←	<p>PREVIOUS WORD: Moves the cursor to beginning of the preceding word, i.e., to the first character to the left of the cursor (in the set [A..Z] or [a..z] or [0..9]) which is preceded by a blank or a special character.</p> <p>For example:</p> <pre>30 IF L < = 0 THEN 20</pre> <p>The cursor is under the letter T. If you press CTRL ←, the cursor will move to 0. Pressing CTRL ← again, it will move to L.</p>
END	<p>END, APPEND: Moves the cursor from its current position to the end of the logical line. Subsequently entered characters are appended to the line.</p>
CTRL END	<p>ERASE TO END OF LINE: Erases from the current cursor position to the end of the logical line, i.e., until the carriage return is found.</p>
INS	<p>SWITCH INSERT/OVERWRITE MODE: Switches into or out of Insert Mode. If Insert Mode is off (Overwrite Mode on), then it turns it on. If Insert Mode is on, then it turns it off (sets Overwrite Mode).</p> <p>Insert mode is also turned off by pressing one of the cursor movement keys or CR.</p> <p>The Insert Mode cursor is a half-height blinking block in Text Mode and is a blinking triangle to the left of each character in Graphics Mode.</p>

GETTING STARTED

KEY	FUNCTION
	<p>Overwrite mode is indicated by a different cursor, which is a slow-blinking under-line. In Insert Mode the character immediately above, together with those following the cursor, move to the right as characters are entered at the current cursor position. As characters disappear off the right side of the screen, they reappear on the left on the following line.</p> <p>When out of Insert Mode, characters typed will replace existing characters on the line.</p>
→	<p>TAB: When out of Insert Mode, pressing → moves the cursor over characters until the next tab stop is reached. Tab stops occur every 8 character positions starting from position 1.</p> <p>For example, given the line below:</p> <pre>20 INPUT "Length"; L</pre> <p>If you press the → key, the cursor will move to the 17th position as shown:</p> <pre>20 INPUT "Length_"; L</pre> <p>When in Insert Mode, pressing → causes blanks to be entered from the current cursor position to the next tab stop. As characters disappear off the right side of the screen, they reappear on the left on the following line.</p>

KEY	FUNCTION
	<p>For example, given the line below:</p> <pre>20 INPUT "Length"; L</pre> <p>Blanks are entered up to the 17th position by pressing the INS key and then the → key.</p> <pre>20 INPUT " Length"; L</pre>
DEL	<p>DELETE CHARACTER: Erases the character located at the current cursor position. All characters which follow the deleted character shift one position left. If a logical line extends beyond one physical line, characters on subsequent lines shift left one position, and the character in the first column of each subsequent line is moved up to the end of the preceding line.</p>
← on Keyboard 1 (or BS on Keyboard 2)	<p>BACKSPACE: Causes the last character typed to be deleted, or deletes the character to the left of the cursor. All characters to the right of and above the cursor shift left one position. Subsequent characters and lines within the current logical line move up as with the DEL key.</p>
CTRL CR	<p>LINE FEED: Causes subsequent text to start automatically on the next screen line.</p>

GETTING STARTED

KEY	FUNCTION
ESC	DELETE LINE: The entire logical line containing the cursor is cleared. The line is not entered for processing. If it is an existing program line, it is not deleted from the program currently in memory.
CTRL BREAK	BREAK: Returns to Command Level, without saving any modifications that were made to the current line being edited. Unlike ESC , it does not delete the line from the screen.

CORRECTING THE CURRENT LINE

All text entered at GW-BASIC Command Level is processed by the Screen Editor. You can therefore use any of the Special Screen Editor Keys described above.

GW-BASIC remains at Command Level after the prompt Ok and until a RUN command is received.

Character Modification

If you make a mistake while entering a line then proceed as follows:

STEP	ACTION
1	<p>You discover the error. For example, suppose you have typed:</p> <pre>RUN "K,PROGR_</pre> <p>when you should have entered:</p> <pre>RUN "A:PROGR_</pre>
2	<p>Use Cursor-Left, or other cursor movement keys, to move the cursor to the appropriate position:</p> <pre>RUN "K,PROGR</pre>
3	<p>Type the correct character over the wrong ones:</p> <pre>RUN "A:PROGR</pre>
4	<p>Move the cursor to the end of the line using Cursor Right or END keys:</p> <pre>RUN "A:PROGR_</pre>
5	<p>Continue typing if the line is not finished:</p> <pre>RUN "A:PROGRAM1" _</pre>
6	<p>Enter CR to pass the line to GW-BASIC. In this case the specified program is loaded from the diskette inserted in drive A and run.</p>

GETTING STARTED

Character Insertion

If you accidentally omit characters in the line you are entering, then proceed as follows:

STEP	ACTION
1	<p>You notice the error.</p> <p>Suppose you entered:</p> <pre>10 FO K = 1 TO _</pre> <p>instead of:</p> <pre>10 FOR K = 1 TO _</pre>
2	<p>Use Cursor-Left, or other cursor movement keys, to move the cursor to the appropriate position:</p> <pre>10 FO _ K = 1 TO</pre>
3	<p>Press INS and type the letter R:</p> <pre>10 FOR _ K = 1 TO</pre> <p>Note that, entering Insert Mode, the cursor becomes a halfheight block.</p>
4	<p>Press INS again to return to Overwrite Mode and Cursor-Right or END to move the cursor to the end of the line:</p> <pre>10 FOR K = 1 TO _</pre>

Character Deletion

If you accidentally type an extra character in the line you are entering, then proceed as follows:

STEP	ACTION
1	<p>You discover the error.</p> <p>For example, suppose you typed:</p> <p>GOTTO_</p> <p>instead of:</p> <p>GOTO_</p>
2	<p>To erase the extra T, press Cursor-Left, or other cursor movement keys, to move the cursor to the appropriate position:</p> <p>GOTTO</p>
3	<p>Press DEL :</p> <p>GOTO_</p>
4	<p>Move the cursor using Cursor-Right:</p> <p>GOTO_</p>
5	<p>Continue typing:</p> <p>GOTO 1000_</p>

GETTING STARTED

Deleting Part of a Line

To erase a line from the current cursor position, press **CTRL END** .

Deleting an Entire Line

To cancel the line you are entering, press **ESC** anywhere in the line. It is not necessary to press **CR** .

MODIFYING PROGRAM LINES

Any line of text beginning with a number (0 to 65529) is considered to be a 'program line'. Suppose you have entered a program, i.e. a sequence of program lines, that you want to modify:

IF you want to...	THEN...
add a new line to your program	enter a valid line number followed by at least one non-blank character, followed by CR .
replace an existing line	enter a line number that matches an existing one, followed by the contents of the new line. The new line will replace the existing one.
delete a line	<p>enter the line number of the line to be deleted, followed by CR.</p> <p>An "Undefined line number" error is returned if an attempt is made to delete a line which does not exist.</p> <p>Note: ESC should not be used to delete program lines, since this erases from the screen only, and not from the program in memory.</p>

IF you want to...	THEN...
delete a group of lines	enter a DELETE command indicating the range of lines to be deleted.
delete the program resident in memory	enter a NEW command.
modify a program line which is already displayed on the screen	move the cursor to the appropriate position (by the cursor movement keys); modify the line using any of the techniques described above to change, delete or insert characters to the line; press CR . to pass the modified line to GW-BASIC.
modify a program line which is not displayed on the screen	<p>use the EDIT command to display the line, or the LIST command to display a group of lines including the line you want to modify, move the cursor to the appropriate position, modify the line, and press CR.</p> <p>Note: You can edit any line as long as it is visible on the screen. Once an immediate line has been sent to the system by pressing CR, there is no way to edit it; this is not the case with program lines, as they may always be recalled for editing to the screen.</p>

Remarks

Changes to a line are recorded when a carriage return is entered while the cursor is somewhere on that line. The carriage return enters all changes for that logical line, and, up to the 255 character line limitation, no matter how many physical lines are included and no matter where the cursor is located on the line.

GETTING STARTED

Note that any modifications you made by using the GW-BASIC Screen Editor only change the program in memory. To store the updated version of your program in a disk file you must use the SAVE command.

USING YOUR SYSTEM AS A CALCULATOR

You can use your Personal Computer as a calculator for quick computation, and debugging purposes.

When you are in GW-BASIC, and the Ok prompt is on the screen, you can enter PRINT (or simply a question mark "?"), followed by any expression, and CR . The expression is evaluated and its value displayed.

You can also enter LET, followed by any variable name, the assignment operator (=), any expression, and CR . The expression is evaluated and its value is assigned to the specified variable. You can use the variable to represent that value in successive computations. The keyword LET is optional, you can begin the line simply using the variable name.

The following table gives some examples.

DISPLAY	COMMENTS
PRINT 3 3 Ok	The constant 3 is displayed.
PRINT 2 + 3 5 Ok	The expression 2 + 3 is evaluated, and its value (5) is displayed.
LET A = 15.21 Ok	The constant 15.21 is assigned to the variable A. You can use A in successive computations to represent this value.

DISPLAY	COMMENTS
?A-1 14.21 Ok	The expression A-1 is evaluated, and its value (14.21) is displayed. Note: ? is equivalent to PRINT.
B = 2.3 Ok	The constant 2.3 is assigned to the variable B. The keyword LET is optional, you may begin with a variable name.
?A*B 34.983 Ok	The expression A*B is evaluated. Its value (34.983) is displayed.
?A*B-40 -5.017002 Ok	The expression A*B-40 is evaluated, and its value (-5.017002) is displayed. Note: If a value is negative, the minus sign is displayed, if a value is positive, no sign is displayed.

ENTERING A PROGRAM

A GW-BASIC program consists of a series of statements. A statement is a complete instruction in GW-BASIC, telling your computer to perform specific operations.

You can enter either one or several statements per line. In the latter case, each statement must be separated from the last by a colon (:).

Each line in a GW-BASIC program begins with a line number: an integer greater than or equal to 0 and less than or equal to 65529 and ends when you press **CR**.

GETTING STARTED

A GW-BASIC line may contain a maximum of 255 characters including the carriage return. The extra characters will be truncated when you enter CR .

When you are in GW-BASIC, and the Ok prompt is on the screen, you can enter a program.

First of all enter:

NEW

This will clear the memory.

Then enter:

```
10 REM RECT1
20 INPUT "Length";L
30 IF L <= 0 THEN 20
40 INPUT "Width";W
50 IF W <= 0 THEN 40
60 LET AREA=L*W
70 PRINT "Area=";AREA;" L=";L;" W=";W
80 GOTO 20
90 END
```

It is conventional to use an interval of 10 between each line number. This allows you to modify the program simply by inserting statements between existing lines.

These statements form a complete program that solves a very simple problem. The problem is to find the area of a rectangle by entering the values of length and width via the keyboard. It has been selected both for its simplicity and to illustrate a variety of the GW-BASIC language features. Other more concise solutions exist.

AUTOMATIC LINE NUMBERING

You can use the AUTO command, to generate a line number automatically each time you press CR. You can exit AUTO mode by pressing CTRL BREAK.

LISTING A PROGRAM

Once a program is in main memory it can be listed. To list your program, enter either the LIST command (the listing will appear on the screen) or, if a printer is connected, the LLIST command (the listing will be printed out).

The LIST and LLIST commands edit your program by converting to upper case letters any keywords, variable names, and function names and to PRINT any question mark (?) used instead of PRINT. Moreover, statements are ordered in ascending line number sequence, even though you may have entered them in a different order.

To list our sample program on the screen enter:

LIST

The screen display is as follows:

LIST

```
10 REM RECT1
20 INPUT "Length";L
30 IF L < =0 THEN 20
40 INPUT "Width";W
50 IF W < =0 THEN 40
60 LET AREA=L*W
70 PRINT "Area="";AREA;" L="";L;" W="";W
80 GOTO 20
90 END
Ok
```

Note that at the end of a listing your system enters command level and displays the Ok prompt; the program can now be edited as required.

SAVING A PROGRAM

A program is kept in memory as long as your computer is switched on. As soon as you turn off your computer or do a system reset, your program is lost. If you want to retain your newly written program for future use, then you must enter a SAVE command to store the program on a disk.

GETTING STARTED

You should save the current program (i.e. the program presently resident in the main memory) on the following occasions:

IF you want to ...	THEN ...
turn off the machine or do a system reset)	save the current program (unless you already have a copy). For example: SAVE "A:RECT1"
enter another program from keyboard	save the current program (unless you already have a copy).
load another program from disk (by entering a LOAD or RUN command)	save the current program (unless you already have a copy).
return to MS-DOS (by entering a SYSTEM command)	save the current program (unless you already have a copy).
replace the old version of your program	save the current program on the same disk, and with the same file name as the old version.
save the current program in ASCII (source) format	specify the A option in the SAVE command. For example: SAVE "A:RECT1", A
protect the current program against any attempt to list, edit or save it again	specify the P option in the SAVE command. For example: SAVE "A:RECT1",P

LOADING A PROGRAM

If the program you want to enter into the main memory resides on a disk, you must issue a LOAD command. LOAD deletes all variables and program lines currently residing in memory, thus before entering a LOAD command you should save the current program if you want to use it again, unless you already have a copy.

To load a program file from a disk, you must specify the drive before the file name, unless the file resides on the default drive. For example:

LOAD "B:ROOT1"

if the program ROOT1 resides on the diskette inserted in drive B.

If you specify the R option, all open data files are kept open and the program is run after it is LOADED. For example:

LOAD "B:ROOT1",R

If you do not specify the R option, LOAD closes all data files.

EXECUTING A PROGRAM

Once a program is in main memory, it can be executed (or "run", as this is frequently called). To tell your system to execute a program, you must enter a RUN command (or a LOAD with the option R).

The RUN command runs the current program, i.e. the program currently in memory, or loads a program from a disk and runs it (if you enter a file specifier after the keyword RUN). For example:

RUN "B:RECT1"

Note that a file specifier is a string expression or, in particular, a string constant. If it is a string constant as in the example above it must be enclosed within quotation marks ("").

If you specify the R option all open data files are kept open, thus you can re-use these files in the new program without having to open them again.

GETTING STARTED

Before entering a RUN *filespec* (or RUN *filespec*,R), save your current program (unless you already have a copy).

GW-BASIC statements are executed in line number sequence, unless a control statement (GOTO, ON...GOTO, IF...GOTO...ELSE, IF...THEN...ELSE, FOR/NEXT, WHILE/WEND) or a subroutine call statement (GOSUB, ON...GOSUB) dictates otherwise.

RUNNING A SAMPLE PROGRAM

Let us run our sample program. Let us suppose it is already in memory, entered through the keyboard or loaded from disk by the LOAD command.

Enter:

LIST

to check that this program is in main memory. The listing will appear on the screen. At the end of the listing, when Ok appears on the screen, enter:

RUN

The following listing is displayed:

LIST

```
10 REM RECT1
20 INPUT "Length";L
30 IF L <= 0 THEN 20
40 INPUT "Width";W
50 IF W <= 0 THEN 40
60 LET AREA=L*W
70 PRINT "Area = ";AREA;" L = ";L;" W = ";W
80 GOTO 20
90 END
```

Ok

RUN

```
Length? 3.5
Width? 4.2
Area = 14.7 L = 3.5 W = 4.2
Length? -7.3
Length? 7.3
Width? 1.3Q
```



```
?Redo from start
Width? 1.32
Area = 9.636 L = 7.3 W = 1.32
Length? CTRL BREAK
Break in 20
Ok
```

Your system begins executing statements sequentially. Because statement 10 is a REM(ark) it is not executed; execution in this case starts with statement 20.

When an INPUT statement is encountered (see statements 20 and 40) program execution is suspended and your system prompts a message indicating that you should enter a value. You could enter for example 3.5 for the length and 4.2 for the width.

Statement 60 calculates the value of AREA. Statement 70 displays the values of AREA, L and W. Statement 80 returns control to statement 20.

If you enter a negative value (e.g. -7.3), for L, statement 20 is executed again, as statement 30 returns control to statement 20 if L is negative or zero.

If you enter a negative value for W, statement 40 is executed again, as statement 50 returns control to statement 40, if W is negative or zero.

If you enter a string value for L or W (e.g. 1.3Q for W) the system displays an error message:

```
?Redo from start
```

and you must re-enter the value.

This program continues to run until you press **CTRL BREAK** to stop execution. Your system displays a "Break in nnnnn" message and returns to Command Level. To resume execution enter:

```
CONT
```

GETTING STARTED

PROGRAM INTERRUPTS

Three types of program interrupts are possible:

- Manual interrupts
- Automatic interrupts
- Programmable interrupts

IF...	THEN...
you press CTRL BREAK , (manual interrupt), or a STOP , or an END statement is executed (programmed interrupt)	<p>the program is interrupted, GW-BASIC enters Command Level and displays Ok. CTRL BREAK and STOP do not close any data files but display a "Break in <i>nnnnn</i>" message. END closes all data files but does not display a "Break in <i>nnnnn</i>" message.</p> <p>In any case you can resume execution by entering a CONT command. You can display program variables (by immediate PRINT or PRINT USING statements) or change their values (by immediate LET or SWAP statements). You can also display program lines by an EDIT or LIST command, and modify them.</p> <p>If you modify lines, you cannot continue execution via a CONT command. You can only rerun the program by entering RUN.</p>
a syntax error is found (automatic interrupt)	the program is interrupted, GW-BASIC displays the error message at the line that caused the error, positioning the cursor under the first digit of the line number.

IF...	THEN...
	<p>You can modify the line, and then rerun the program by entering RUN. You cannot continue execution by entering CONT.</p> <p>If you want to examine the contents of some variables before making any modifications you should press CTRL BREAK to return to Command Level. After examining the contents of the variables you can edit the line and rerun the program.</p> <p>For example:</p> <pre> 10 A = 2\$6 RUN ?Syntax Error in 10 Ok 10 A = 2\$6 _</pre>
<p>an error (other than a syntax error) is found (automatic interrupt)</p>	<p>the program is interrupted, GW-BASIC displays the error message, enters Command Level and displays Ok.</p> <p>You can either display program variables or display program lines by an EDIT or LIST command, and then modify them.</p> <p>You cannot continue execution by entering a CONT command, but you can rerun the program by entering RUN.</p> <p>For example, running a program which contains:</p> <pre> 100 FOR K=</pre> <p>will cause:</p> <pre> Missing operand in 100 Ok</pre>

GETTING STARTED

IF...	THEN...
an error occurs and the error trapping is enabled (programmed interrupt)	<p>program execution is transferred to the line specified by the ON ERROR statement.</p> <p>An error trapping routine should check for all the particular errors that the user wishes to recover from, and should specify the course of action to be taken in each case.</p> <p>This either involves correcting the error, and resuming execution at a specified statement or, returning to Command Level.</p>
an event occurs and the corresponding event trapping is enabled (programmed interrupt)	<p>program execution is transferred to the line specified by the ON <i>event</i> GOSUB statement. An event trapping routine should specify the course of action to be taken in this case. The events that can be trapped are: receipt of characters from a communications port; detection of certain keystrokes; time passage; or emptying of the background music buffer. See Chapter 6 ("Event Trapping") for further details.</p>

Example

```
10 ON ERROR GOTO 100
20 INPUT "WHAT IS YOUR BET"; B
30 IF B > 5000 THEN ERROR 200
.
.
100 IF ERR = 200 THEN PRINT "HOUSE LIMIT IS $5000"
110 IF ERL = 30 THEN RESUME 20
120 ON ERROR GOTO 0
```


If you enter a value of $B > 5000$, the message:

HOUSE LIMIT IS \$5000

is displayed and execution resumes at line 20.

Statement 10 (ON ERROR GOTO 100) both enables error trapping and transfers control to line 100 if an error occurs. The statement ERROR 200 at line 30 generates the user defined error 200. When this statement is executed control is transferred to line 100.

At statement 120, ON ERROR GOTO 0 disables error trapping, thus if any error is encountered other than error 200 the standard error message will be displayed.

Note the meaning of the special built-in functions ERR and ERL. The former contains the error code, the latter the line number of the line where the error occurs.

3. CONSTANTS, VARIABLES AND EXPRESSIONS

ABOUT THIS CHAPTER

This chapter illustrates the principal elements that may be entered in a GW-BASIC line: constants, variables, expressions and operators.

CONTENTS

CONSTANTS	3-1	INTEGER DIVISION AND MODULUS ARITHMETIC	3-10
SINGLE AND DOUBLE PRECISION FORM FOR NUMERIC CONSTANTS	3-2	OVERFLOW	3-10
VARIABLES	3-3	RELATIONAL OPERATORS	3-11
VARIABLES NAMES AND DECLARATION CHARACTERS	3-3	LOGICAL OPERATORS	3-12
ARRAY VARIABLES	3-5	FUNCTIONAL OPERATORS	3-17
MEMORY REQUIREMENTS	3-5	STRING OPERATORS	3-17
TYPE CONVERSION	3-6		
EXPRESSIONS AND OPERATORS	3-8		
ARITHMETIC OPERATORS	3-8		

CONSTANTS, VARIABLES AND EXPRESSIONS

CONSTANTS

Constants are the values that GW-BASIC uses during program execution. There are two types of constant: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks.

Examples:

```
"READY"  
"$80"  
"acceleration rate"
```

Numeric constants are positive or negative numbers. GW-BASIC numeric constants cannot contain commas. There are five types of numeric constant:

1. Integer constants
Whole numbers between -32768 and 32767. Integer constants do not contain decimal points.
2. Fixed-point constants
Positive or negative real numbers, i.e., numbers that contain decimal points.
3. Floating-point constants
Positive or negative numbers represented in exponential form (similar to scientific notation). A floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The range for floating-point constants is $10E-38$ to $10E+38$.

Examples:

```
235.988E-7 is equivalent to .0000235988  
2359E6 is equivalent to 2359000000
```

(Double precision floating-point constants are denoted by the letter D instead of E. See later in this chapter.)

4. Hex constants
Hexadecimal numbers denoted by the prefix &H.

Examples:

&H76
&H32F
&HFFAA

5. Octal constants
Octal numbers denoted by the prefix &O or &.

Examples:

&O347
&1234

SINGLE AND DOUBLE PRECISION FORM FOR NUMERIC CONSTANTS

Numeric constants may be either single precision or double precision numbers. Single precision numeric constants are stored with 7 digits of precision, and printed with up to 7 digits of precision. Double precision numeric constants are stored with 16 digits of precision and printed with up to 16 digits.

A single precision constant is any numeric constant that has one of the following characteristics:

1. Seven or fewer digits.
2. Exponential form using E.
3. A trailing exclamation point (!).

Examples:

46.8
-1.09E-06
3489.0
22.5!

CONSTANTS, VARIABLES AND EXPRESSIONS

A double precision constant is any numeric constant that has one of the following characteristics:

1. Eight or more digits.
2. Exponential form using D.
3. A trailing number sign (#).

Examples:

```
345692811  
-1.09432D-06  
3489.0 #  
7654321.1234
```

VARIABLES

Variables are names used to represent values used in a GW-BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

VARIABLE NAMES AND DECLARATION CHARACTERS

GW-BASIC variable names may be any length. Up to 40 characters are significant. Variable names can contain letters, numbers, and the decimal point. However, the first character must be a letter. Special type declaration characters are also allowed (see below).

A variable name may not be a reserved word, but embedded reserved words are allowed. Reserved words include all GW-BASIC commands, statements, function names, and operator names. If a variable begins with FN, it is assumed to be a call to a user-defined function.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign (\$) as the last character. For example: A\$ = "SALES REPORT". The dollar sign is a variable type declaration character; that is, it "declares" that the variable will represent a string.

Numeric variable names may declare integer, single precision, or double precision values. The type declaration characters for these variable names are as follows:

% Integer variable

! Single precision variable

Double precision variable

The default type for a numeric variable name is single precision.

Examples of GW-BASIC variable names:

PI#	Declares a double precision value.
MINIMUM!	Declares a single precision value.
LIMIT%	Declares an integer value.
N\$	Declares a string value.
ABC	Declares a single precision value.

There is a second method by which variable types may be declared. The GW-BASIC statements DEFINT, DEFSTR, DEFSNG, and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in Chapter 8.

CONSTANTS, VARIABLES AND EXPRESSIONS

ARRAY VARIABLES

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32,767. Both these values are also limited by the memory size of your system.

In the interest of brevity, whenever a specific parameter is referred to as a variable in the syntax of a statement or command, no distinction is made as to whether it is a simple variable or an array element.

MEMORY REQUIREMENTS

The number of bytes required by strings, variables and arrays is listed below.

Variable Type	Bytes
Integer	2
Single Precision	4
Double Precision	8
Array Type	Bytes
Integer	2 per element
Single Precision	4 per element
Double Precision	8 per element

Strings

3 bytes overhead plus the present contents of the string.

TYPE CONVERSION

When necessary, GW-BASIC will convert a numeric constant from one type to another. The following rules and examples should be observed.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)

Example:

```
10 A% = 23.42
20 PRINT A%
RUN
23
```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

Example:

```
10 D# = 6# / 7
20 PRINT D#
RUN
.8571428571428571
```

The arithmetic is performed in double precision and the result is returned in D# as a double precision value.

Example:

```
10 D = 6# / 7
20 PRINT D
RUN
.8571429
```

The arithmetic is performed in double precision and the result is returned to D (single precision variable), rounded, and printed as a single precision value.

CONSTANTS, VARIABLES AND EXPRESSIONS

3. Logical operators convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs. A full description of Logical Operators follows later in this chapter.
4. When a floating-point value is converted to an integer, the fractional portion is rounded.

Example:

```
10 C% = 55.88
20 PRINT C%
RUN
56
```

5. If a double precision variable is assigned a single precision value, only the first seven digits (rounded) of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than $6.3E-8$ times the original single precision value.

Example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
2.04 2.039999961853027
```

EXPRESSIONS AND OPERATORS

An expression may be a string or numeric constant, a variable, or a combination of constants and variables with operators. An expression always produces a single value.

Operators perform mathematical or logical operations on values. The GW-BASIC operators may be divided into four categories:

1. Arithmetic
2. Relational
3. Logical
4. Functional

Each category is described in the following subsections.

ARITHMETIC OPERATORS

The arithmetic operators, in order of precedence, are as follows:

OPERATOR	OPERATION	SAMPLE EXPRESSION
\wedge	Exponentiation	$X \wedge Y$
$-$	Negation	$-X$
$*$, $/$	Multiplication, Floatingpoint Division	$X * Y$ X / Y
\backslash	Integer Division	$X \backslash Y$
MOD	Modulus Arithmetic	$X \text{ MOD } Y$
$+$, $-$	Addition, Subtraction	$X + Y$, $X - Y$

Tab. 3-1 Arithmetic Operators

CONSTANTS, VARIABLES AND EXPRESSIONS

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Within the parentheses, the usual order of operations is maintained.

Some sample algebraic expressions follow, together with their GW-BASIC counterparts.

Algebraic Expression

GW-BASIC Expression

$$X + 2Y$$

$$X + Y * 2$$

$$X - \frac{Y}{Z}$$

$$X - Y / Z$$

$$\frac{XY}{Z}$$

$$X * Y / Z$$

$$\frac{X + Y}{Z}$$

$$(X + Y) / Z$$

$$(X^2)^Y$$

$$(X \wedge 2) \wedge Y$$

$$X^{Y^Z}$$

$$X \wedge (Y \wedge Z)$$

$$X(-Y)$$

$$X * (-Y)$$

Note

Two consecutive operators must be separated by parentheses, as shown in the $X * (-Y)$ example.

INTEGER DIVISION AND MODULUS ARITHMETIC

Two additional operators are available in GW-BASIC: integer division and modulus arithmetic.

Integer division is denoted by the backslash (`\`). The operands are rounded to integers before the division is performed, and the quotient is truncated to an integer. The operands must be within the range -32768 to 32767.

Example

10\4 is 2
25.68\6.99 is 3

Integer division follows multiplication and floating-point division in order of precedence.

Modulus arithmetic is denoted by the operator MOD. Modulus arithmetic yields the integer value that is the remainder of an integer division.

Example

10.4 MOD 4 is 2 ($10/4 = 2$ with a remainder 2)
25.68 MOD 6.99 is 5 ($26/7 = 3$ with a remainder 5)

Modulus arithmetic follows integer division in order of precedence.

OVERFLOW

If, during the evaluation of an expression, division by zero is encountered, the "Division by zero" error message is displayed, machine infinity (the largest number that can be represented in floating-point format) with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation operator results in zero being raised to a negative power, the "Division by zero" error message again is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

CONSTANTS, VARIABLES AND EXPRESSIONS

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

RELATIONAL OPERATORS

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then be used to make a decision regarding program flow.

The relational operators are:

OPERATOR	RELATION TESTED	EXAMPLE
=	Equality	$X = Y$
< > (or > <)	Inequality	$X < > Y$
<	Less than	$X < Y$
>	Greater than	$X > Y$
< = (or = <)	Less than or equal to	$X < = Y$
> = (or = >)	Greater than or equal to	$X > = Y$

Tab. 3-2 Relational Operators

(The equal sign is also used to assign a value to a variable. See the "LET" Statement in Chapter 8.)

When arithmetic and relational operators are combined in one expression, the arithmetic operation is always performed first. For example, the expression:

$$X + Y < (T-1)/Z$$

is true if the value of X plus Y is less than the value of (T-1)/Z.

More examples:

```
IF SIN(X) < 0 GOTO 1000  
IF I MOD J < > 0 THEN K = K + 1
```

LOGICAL OPERATORS

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in Figure 3-3.

The operators are listed in order of precedence.

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

Fig. 3-3 GW-BASIC Logical Operators Truth Table

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

Fig. 3-3 GW-BASIC Logical Operators Truth Table (cont.)

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a subsequent decision.

Examples

```
IF D < 200 AND F < 4 THEN 80
IF I > 10 OR K < 0 THEN 50
IF NOT P THEN 100
```

CONSTANTS, VARIABLES AND EXPRESSIONS

Logical operators work by converting their operands to 16-bit, signed, two's complement integers in the range -32768 to 32767, (if the operands are not in this range, an error results). If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers bit-by-bit, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16 = 16 the result is 16, as
63 = binary 111111
16 = binary 010000
 010000

15 AND 14 = 14 the result is 14, as
15 = binary 1111
14 = binary 1110
 1110

-1 AND 8 = 8 the result is 8, as
-1 = binary 1111111111111111
8 = binary 0000000000001000
 0000000000001000

4 OR 2 = 6 the result is 6, as
4 = binary 100
2 = binary 10
 110

10 OR 10 = 10 the result is 10, as
10 = binary 1010
10 = binary 1010
 1010

-1 OR -2 = -1

the result is -1, as

-1 = binary 1111111111111111

-2 = binary 1111111111111110
1111111111111111

NOT X = -(X + 1)

The two's complement of any integer is the bit complement plus one.

1 1 1 1 1 1 0 0 1 1 0 0 1 1 0 0 0 original value (negative)

inverting all the bits:

0 0 0 0 0 1 1 0 0 1 1 0 0 1 1 1 inverted value

adding 1:

0 0 0 0 0 1 1 0 0 1 1 0 1 0 0 0 absolute value
(inverted + 1)

so the value in decimal of the given pattern is:

1640

CONSTANTS, VARIABLES AND EXPRESSIONS

FUNCTIONAL OPERATORS

When a function is used in an expression, it calls a predetermined operation that is to be performed on an operand. GW-BASIC has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All GW-BASIC intrinsic functions are described in Chapter 8.

GW-BASIC also allows "user-defined" functions that are written by the programmer. (See "DEF FN" Statement in Chapter 8.)

STRING OPERATORS

Strings may be concatenated by using +.

Example

```
10 A$ = "FILE" : B$ = "NAME"  
20 PRINT A$ + B$  
30 PRINT "NEW " + A$ + B$  
RUN  
FILENAME  
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

= <> < > < = (or = <) > = (or = >)

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If during string comparison the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant.

Examples

```
"AA"<"AB"  
"FILENAME"="FILENAME"  
"X&">"X#"  
"CL ">"CL"  
"kg">"KG"  
"SMYTH"<"SMYTHE"  
B $< "9/12/78"      where B$="8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

4. DISK FILE HANDLING

ABOUT THIS CHAPTER

This chapter describes how to create and manage disk files and gives a complete description of their features and functions.

CONTENTS

INTRODUCTION	4-1	SEQUENTIAL FILES	4-10
DEVICE INDEPENDENT INPUT/OUTPUT	4-1	CREATING A SEQUENTIAL FILE	4-10
HOW MS-DOS KEEPS TRACK OF YOUR FILES	4-1	ACCESSING A SEQUENTIAL FILE	4-12
FILE NUMBERS	4-2	ADDING DATA TO A SEQUENTIAL FILE	4-13
NAMING FILES	4-2	RANDOM ACCESS FILES	4-13
NAMING DEVICES	4-3	CREATING A RANDOM ACCESS FILE	4-14
DIRECTORY PATHS	4-4	ACCESSING A RANDOM ACCESS FILE	4-15
CURRENT DIRECTORY	4-7	FILE LOCKING	4-19
COMMANDS FOR PROGRAM FILES	4-7		
PROTECTED FILES	4-9		
DISK DATA FILES - SEQUENTIAL AND RANDOM ACCESS	4-9		

DISK FILE HANDLING

INTRODUCTION

Disk input and output procedures for the GW-BASIC user are examined in this chapter. If you are new to GW-BASIC or if you are getting disk-related errors, read through these procedures and program examples to make sure you are using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to the filenaming conventions for your operating system. The MS-DOS operating system will append a default extension .BAS to the filename given in a SAVE, RUN, MERGE or LOAD command.

DEVICE INDEPENDENT INPUT/OUTPUT

GW-BASIC provides device-independent input/output; using device independent I/O means that the syntax for access is the same for any device.

The following statements, commands, and functions support device-independent I/O:

BLOAD	LOF
BSAVE	MERGE
CHAIN	OPEN
CLOSE	POS
EOF	PRINT #
GET	PRINT # USING
INPUT #	PUT
INPUT\$	RUN
LINE INPUT #	SAVE
LIST	WIDTH
LOAD	WRITE #
LOC	

HOW MS-DOS KEEPS TRACK OF YOUR FILES

A file is a collection of records. The names of files are kept in directories on disk or diskette. These directories also contain information on the size of the files, their location on the disk, and the dates that they were created and updated. The directory you are working in is called your current directory.

An additional system area is called the File Allocation Table. It keeps track of the location of your files on the disk. It also allocates the free space on your disks so that you can create new files.

These two system areas, the directories and the File Allocation Table, enable MS-DOS to recognize and organize the files on your disks. The File Allocation Table is created onto a new disk when you format it with the MS-DOS FORMAT command, and one empty directory is created, known as the "root" directory.

To use the information you must OPEN the file to tell GW-BASIC where the information is. You may then use the file for input and/or output.

FILE NUMBERS

The maximum number of files that can be simultaneously opened is set by the /F: option in the GWBASIC command. A file number is associated with a file when the file is opened and it is used by any subsequent I/O statement to refer to the file (see the GWBASIC command and the OPEN statement in Chapter 8).

NAMING FILES

File specifications follow MS-DOS naming conventions. The *filespec* is a string expression with the following format:

[*device:*] *filename*

All file specifications may begin with a device specification such as A: or B: or COM1: or LPT1:. If no device is specified, the current drive is assumed.

A *filename* can comprise:

- one to eight characters (for legal characters see below). For example NEWFILE.
- one to eight characters, followed by a period (.) and a one to three character file name extension. For example NEWFILE.EXE.

DISK FILE HANDLING

A *filename* may be made up of any of the following characters:

A-Z	0-9	\$	&	#	~
%	'	()	-	_
@	^	{	}	!	

Alphabetic characters within the file name can be entered in upper or lower case, but MS-DOS will translate lower case letters into upper case. The extension .BAS is supplied if no extension is given, but NAME and KILL do not follow this rule: they do not supply any extension.

Note

File specification for communications devices is slightly different. The filename is replaced with a list of options specifying such things as line speed. Refer to the OPEN COM statement in Chapter 8 for details.

Remember that if you use a string constant for the file specification, you must enclose it in quotation marks. The only exception to this rule is the GWBASIC command, where a file specifier is a string constant not included in quotation marks.

For example:

LOAD "B:ARSENAL.RED"

For example:

GWBASIC PAYROLL

NAMING DEVICES

The device name is a string of four characters or less followed by a colon (:), and may be one of the following:

A: first diskette drive	(Any access mode)
B: second diskette drive	(Any access mode)
C: first hard disk drive	(Any access mode)

D:	second hard disk drive	(Any access mode)
KYBD:	keyboard	(Input only)
SCRN:	screen	(Output only)
LPT1:	first printer	(Output or random)
LPT2:	second printer	(Output or random)
LPT3:	third printer	(Output or random)
COM1:	RS232 Port 1	(Input and Output)
COM2:	RS232 Port 2	(Input and Output)

DIRECTORY PATHS

With GW-BASIC the user can organize a disk in such a manner that files that are not part of his current task do not interfere with that task.

Previously, only a single directory was supported that contained all the files on a disk. MS-DOS extends this concept to allow a directory to contain both files and directories and to introduce the notion of the current directory.

To specify a file, the user could use one of two methods, either specify a path from the root directory to the file, or specify a path from the current directory to the file. A "Directory Path" (or *pathname*) is a series of directory names separated by '\' and ending with a file name. A *pathname* that starts at the root begins with the '\'.

There are two special directory entries in each directory, denoted by '.' and '..'. They specify the directory itself ('.') or the parent of the directory ('..'). The root directory's parent is itself.

DISK FILE HANDLING

Let us take a hypothetical example.

In a particular business, both sales and accounting share a computer with a large disk and the individual employees use it for preparation of reports and maintaining accounting information. One would naturally view the organization of files on the disk in this fashion:

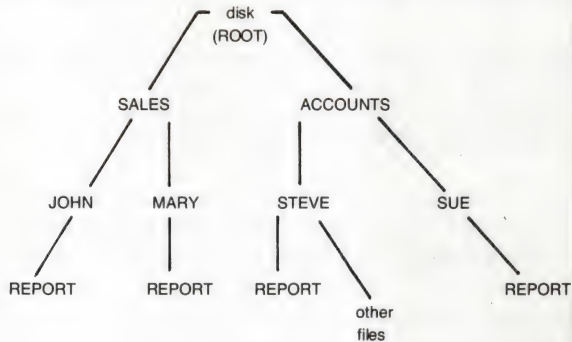


Fig. 4-1 Disk File Organization

Using a directory structure like the hierarchy above, and assuming that the current directory is directory JOHN, to reference the REPORT under JOHN, the following are equivalent:

REPORT

\SALES\JOHN\REPORT

To refer to the REPORT under MARY, supposing that JOHN is still the current directory, the following are equivalent:

..\MARY\REPORT

\SALES\MARY\REPORT

To refer to the REPORT under SUE, supposing that JOHN is still the current directory, the following are equivalent:

```
..\..\ACCOUNTS\SUE\REPORT
```

```
\ACCOUNTS\SUE\REPORT
```

There is no restriction on the depth of a tree (the length of the longest path from root to leaf).

The root directory will have a fixed maximum number of entries for a diskette. There is no limit to the number of files and/or subdirectories in the root directory of a hard disk, other than the size of the MS-DOS partition (see the "MS-DOS User Guide").

Other "sub-directories" can also be accessed via the root directory, and these in turn can branch off to further files and sub-directories. The only limit being the space available on the disk.

Each directory can also contain file and directory names that also appear in other directories.

Pathnames can be used for the following commands:

BLOAD	GW BASIC(*)	NAME
BSAVE	KILL	OPEN
CHAIN	LOAD	RMDIR
CHDIR	MERGE	RUN
FILES	MKDIR	SAVE

(*) Used to initialize GW-BASIC. This is an MS-DOS command (not a GW-BASIC command).

A *pathname* may be considered as an extension of *filespec* and is a string expression of the form:

```
[ device: ][ \ ][ directory ][ \directory ]...[ \ ] filename
```

or

```
[ device: ][ \ ][ directory ][ \directory ]...[ directory ] [ \ ]
```

All characters that are valid for a filename are also valid for a directory name.

DISK FILE HANDLING

Examples (supposing JOHN is the current directory):

B:\SALES\MARY\REPORT
B:..\MARY\REPORT is equivalent

Note

A *pathname* may not contain more than 63 characters. Pathnames longer than 63 characters will give a "Bad Filename" error.

Specifying a *pathname* where only a *filespec* is legal, or placing a *device* other than at the beginning of the *pathname* will result in a "Bad Filename" error.

If you use a string constant for the *pathname*, you must enclose it in quotation marks. The GWBASIC command only specifies pathnames as literal strings not included in quotation marks.

CURRENT DIRECTORY

If you enter a *filename*, in a GW-BASIC statement or command, without specifying a directory, the "current directory" is searched. A single directory is created on a disk when it is formatted. That directory is called the "root" directory, and it is the current directory, initially. You can create other directories by entering the MKDIR command, or remove directories by entering the RMDIR command. The CHDIR command allows you to change the current directory.

If a *pathname* begins with a backslash (\), GW-BASIC starts its search from the "root"; otherwise it starts its search from the current directory. The *pathname* you specify can be either a sequence of directory names starting with the "root", or with the current directory. If the file belongs to the current directory you only need to specify the file.

COMMANDS FOR PROGRAM FILES

The following list reviews the commands and statements used in program file manipulation.

With GW-BASIC the asterisk (*) and question mark (?) can be used as wild cards with the FILES and KILL commands.

SYNTAX	MEANING
SAVE <i>filespec</i> [{A P}] or SAVE <i>pathname</i> [{A P}]	Writes to disk the program that currently resides in memory. Option A writes the program as a series of ASCII characters (otherwise, GW-BASIC uses a compressed binary format); option P writes the program in a protected form. (See "Protected Files" in this chapter).
LOAD <i>filespec</i> [,R] or LOAD <i>pathname</i> [,R]	Loads the program from disk into memory. Option R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before loading. If R is included, however, open data files are kept open. Thus, programs may be chained or loaded in sections and access the same data files. LOAD <i>filespec</i> , R and RUN <i>filespec</i> , R are equivalent.
RUN <i>filespec</i> [,R] or RUN <i>pathname</i> [,R]	RUN <i>filespec</i> loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included however, all open data files are kept open.
MERGE <i>filespec</i> or MERGE <i>pathname</i>	Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk merge with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and GW-BASIC returns to command level.

DISK FILE HANDLING

SYNTAX	MEANING
KILL <i>filespec</i> or KILL <i>pathname</i>	Deletes the file from the disk. The file may be a program file, or a sequential or random access data file.
NAME { <i>filespec</i> <i>pathname</i> } AS <i>filename</i>	Changes the name of a disk file. NAME may be used with any disk file.

PROTECTED FILES

If you want to save a program in an encoded binary format, use the P option with the SAVE command. For example:

SAVE "MYPROG",P

Note

Because a program saved in this manner cannot be listed or edited, you may want to save an unprotected copy of the program for this purpose.

DISK DATA FILES - SEQUENTIAL AND RANDOM ACCESS

Two types of disk data files can be created and accessed by a GW-BASIC program:

- sequential files
- random access files

SEQUENTIAL FILES

Sequential files are easier to create than random access files but limit flexibility and speed when accessing the data. The data written to a sequential file is in the form of ASCII characters which are loaded and stored, one item after another (sequentially), in the order they are sent.

The statements and functions used with sequential files are as follows:

```
CLOSE
EOF
INPUT$
INPUT #
LINE INPUT #
LOC
LOCK
LOF
OPEN
PRINT #
PRINT # USING
UNLOCK
WRITE #
```

See Chapter 8 of this manual for more information on these statements and functions.

CREATING A SEQUENTIAL FILE

The following program steps are required to create a sequential file and access the data in the file:

1. OPEN the file in "O" mode: *= output mode*
OPEN "O", #1, "DATA"
2. Write data to the file using the WRITE # statement (you may use the PRINT # statement instead):

```
WRITE #1,A$;B$;C$
```

DISK FILE HANDLING

3. To access the data in the file, you must CLOSE the file and reOPEN it in "I" mode:

```
CLOSE #1  
OPEN "I", #1, "DATA"
```

4. Use the INPUT # statement to read data from the sequential file to the program:

```
INPUT #1, X$, Y$, Z$
```

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement:

```
PRINT #1, USING "###.###"; A, B, C, D
```

could be used to write numeric data to disk without explicit delimiters. The comma (,) at the end of the format string serves to separate each item in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was opened. For example:

```
100 IF LOC(1) > 50 THEN STOP
```

would end program execution if more than 50 sectors had been written to, or read from, file #1 since it was opened.

Program 1 is a short program that creates a sequential file, named "DATA," from information you input at the terminal.

```
10 OPEN "O", #1, "DATA"  
20 INPUT "NAME"; N$  
25 IF N$ = "DONE" THEN END  
30 INPUT "DEPARTMENT"; D$  
40 INPUT "DATE HIRED"; H$  
50 PRINT #1, N$, " ", D$, " ", H$  
60 PRINT: GOTO 20
```

Fig. 4-2 Program 1 - Create a Sequential Data File

RUN

NAME? **MICKEY MOUSE**
DEPARTMENT? **AUDIO/VISUAL AIDS**
DATE HIRED? **01/12/72**

NAME? **SHERLOCK HOLMES**
DEPARTMENT? **RESEARCH**
DATE HIRED? **12/03/65**

NAME? **EBENEEZER SCROOGE**
DEPARTMENT? **ACCOUNTING**
DATE HIRED? **04/27/78**

NAME? **SUPER MAN**
DEPARTMENT? **SECURITY**
DATE HIRED? **08/16/78**

NAME? etc.

Fig. 4-2 Program 1 - Create a Sequential Data File (cont.)

ACCESSING A SEQUENTIAL FILE

Program 2 accesses the file "DATA" that was created in Program 1 and displays the name of everyone hired in 1978.

```
10 OPEN "I", #1, "DATA"
20 INPUT #1, N$, D$, H$
30 IF RIGHT$(H$, 2) = "78" THEN PRINT N$
40 GOTO 20
```

= input mode

RUN

EBENEEZER SCROOGE
SUPER MAN
Input past end in 20
Ok

Fig. 4-3 Program 2 - Access a Sequential File

Program 2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. This error can be avoided, however, by inserting an additional line (line 15 shown below) which uses the EOF function to test for end-of-file.

DISK FILE HANDLING

```
15 IF EOF(1) THEN END
```

Then change line 40 to GOTO 15.

ADDING DATA TO A SEQUENTIAL FILE

As soon as a sequential file is opened on disk in "O" mode, its current contents are destroyed. In order to add more data to the file it is necessary to use the OPEN statement with the APPEND mode.

RANDOM ACCESS FILES

Creating and accessing random access files requires more program steps than with sequential files, but there are advantages to using random access files. Random access files require less room on the disk, because GW-BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random access files is that data can be accessed at random, i.e., anywhere on the disk. It is not necessary to read through all the information on disk with random access files, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record has an identification number.

The statements and functions that are used with random access files are:

- CLOSE
- CVD
- CVI
- CVS
- FIELD
- GET
- LOC
- LOCK
- LOF
- LSET
- MKD\$
- MKI\$
- MKS\$
- OPEN
- PUT
- RSET
- UNLOCK

CREATING A RANDOM ACCESS FILE

Creation of a random access file requires the following program steps.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

```
OPEN "R", #1, "FILE", 32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.

```
FIELD# 1, 20 AS N$, 4 AS A$, 8 AS P$
```

3. Use the LSET command to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions. MKI\$ converts an integer into a string, MKS\$ converts a single precision number into a string, and MKD\$ converts a double precision number into a string.

```
LSET N$ = X$  
LSET A$ = MKS$(AMT)  
LSET P$ = TEL$
```

4. Write the data from the buffer to the disk using the PUT-statement.

```
PUT #1, CODE%
```

The LOC function, with random access files, returns the "current record number". The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1) > 50 THEN END
```

ends program execution if the current record number in file # 1 is higher than 50.

Program 3 writes information that is input at the terminal to a random access file.

DISK FILE HANDLING

```
10 OPEN "R", #1, "FILE", 32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE"; CODE%
40 INPUT "NAME"; X$
50 INPUT "AMOUNT"; AMT
60 INPUT "PHONE"; TEL$: PRINT
70 LSET N$ = X$
80 LSET A$ = MKS$(AMT)
90 LSET P$ = TEL$
100 PUT #1, CODE%
110 GOTO 30
```

Fig. 4-4 Program 3 - Create a Random Access File

Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

Warning

Do not use a FIELDed string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of into the random access file buffer.

ACCESSING A RANDOM ACCESS FILE

Creation of a random access file requires the following steps.

1. OPEN the file in "R" mode:

```
OPEN "R", #1, "FILE", 32
```

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file:

```
FIELD #1,20 AS N$, 4 AS A$,8 AS P$
```

Note

In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer:

```
GET #1, CODE%
```

4. The data in the buffer may now be accessed by the program. Numeric values that are read in from a random file buffer must be converted from strings back into numbers using the "convert" functions. CVI converts a 2-byte string to an integer, CVS converts a 4-byte string to a single precision number, and CVD converts an 8-byte string to a double precision number:

```
PRINT N$
```

```
PRINT CVS(A$)
```

Program 4 accesses the random access file "FILE" that was created in Program 3. When the two-digit code is entered at the terminal, the information associated with that code is read from the file and displayed.

```
10 OPEN "R", #1, "FILE", 32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE"; CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$# # #.# #"; CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

Fig. 4-5 Program 4 - Access a Random Access File

DISK FILE HANDLING

Program 5 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900 through 960 initialize the data file by writing CHR\$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 140 through 210 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

```
120 OPEN"R",#1,"INVEN.DAT",39
130 FIELD#1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 AS P$
140 PRINT:PRINT "FUNCTIONS":PRINT
150 PRINT 1,"INITIALIZE FILE"
160 PRINT 2,"CREATE A NEW ENTRY"
170 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
180 PRINT 4,"ADD TO STOCK"
190 PRINT 5,"SUBTRACT FROM STOCK"
200 PRINT 6,"DISPLAY ALL ITEMS BELOW REORDER LEVEL"
210 PRINT:PRINT:INPUT"FUNCTION";FUNCTION
220 IF (FUNCTION<1)OR(FUNCTION>6) THEN PRINT "BAD
    FUNCTION NUMBER":GOTO 140
230 ON FUNCTION GOSUB 900,250,390,480,560,680
240 GOTO 210
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT"OVERWRITE";A$: IF
    A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MK$(P)
370 PUT #1,PART%
```

Fig. 4-6 Program 5 - Inventory

```

380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$#.#";CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$:INPUT "QUANTITY TO ADD ";A%
520 Q% = CVI(Q$) + A%
530 LSET Q$ = MKI$(Q%)
540 PUT #1,PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$) = 255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q% = CVI(Q$)
620 IF (Q%-S%) < 0 THEN PRINT "ONLY";Q%," IN STOCK":GOTO
    600
630 Q% = Q%-S%
640 IF Q% = < CVI(R$) THEN PRINT "QUANTITY NOW";Q%; "
    REORDER LEVEL";CVI(R$)
650 LSET Q$ = MKI$(Q%)
660 PUT #1,PART%
670 RETURN
680 REM DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I = 1 TO 100
710 GET #1,I
720 IF CVI(Q$) < CVI(R$) THEN PRINT D$;" QUANTITY"; CVI(Q$)
    TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%

```

Fig. 4-6 Program 5 - Inventory (cont.)

DISK FILE HANDLING

```
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART
    NUMBER": GOTO 840 ELSE GET #1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$ = CHR$(255)
930 FOR I = 1 TO 100
940 PUT #1,I
950 NEXT I
960 RETURN
```

Fig. 4-6 Program 5 - Inventory (cont.)

FILE LOCKING

There are three GW-BASIC statements that enable the user to control the access rights of disk files: OPEN, LOCK and UNLOCK. The LOCK and UNLOCK statements and the *access* parameter of OPEN can only be used if MS-NET is installed with MS-DOS release 3.1 or later. All three statements are fully described in Chapter 8.

5. GRAPHICS

ABOUT THIS CHAPTER

GW-BASIC provides, under MS-DOS, a complete range of graphic features. The design of different shapes, using a range of colors holds a sizable amount of possibilities. The screen can be subdivided into rectangular sections, so that several different areas can be viewed at the same time.

This chapter introduces screen graphics including screen modes, color use, coordinates, the statements VIEW and WINDOW, and other graphics statements.

CONTENTS

SELECTING THE GRAPHICS ENVIRONMENT	5-1	COORDINATES	5-10
		ANIMATION TECHNIQUES	5-11
TEXT MODE	5-2		
GRAPHICS MODE	5-4		
MEDIUM RESOLUTION MODE	5-5		
HIGH RESOLUTION MODE	5-6		
SUPER RESOLUTION MODE	5-7		
WINDOWS AND VIEWPORTS	5-8		
TRANSFORMATION USING WINDOW AND VIEW	5-9		

GRAPHICS

SELECTING THE GRAPHICS ENVIRONMENT

The GW-BASIC language provides a set of graphics statements and functions which permit you to draw lines and points on the video using various colors. You must select the graphics environment in order to use these statements.

Upon initialization, the system is in Text Mode (SCREEN 0); you select the graphics environment using the SCREEN statement.

There are three different graphics modes you can select:

- Medium Resolution Mode (by entering SCREEN 1)
- High Resolution Mode (by entering SCREEN 2)
- Super Resolution Mode (by entering SCREEN 3)

They differ in the number of points displayed and in the number of colors allowed.

The SCREEN statement also allows you to select the "active" and the "visual" page in text mode (using the *apage* and *vpage* parameters). The active page, is the page written to by subsequent output statements to the screen; the visual page is the one displayed on the screen.

The SCREEN statement must precede any I/O statements to the screen. The system assumes SCREEN 0,0,0,0 by default; this selects 80 columns Text Mode, and one display page.

You can also use more than one SCREEN statement to define different screen attributes for different sections of your program.

You can also change from one graphics mode to another by the WIDTH statement. The WIDTH statement allows you to set the screen width (in Text Mode), to select a text "window", or change mode (in one of the graphics modes).

TEXT MODE (SCREEN 0)

In Text Mode you can display text, i.e. letters, numbers, and all special characters of the GW-BASIC character set. You can set the character foreground and background color using the COLOR (Text) statement. This statement also allows you to create blinking, reverse image, invisible, highlighted, and underscore characters.

Characters are displayed in horizontal lines from top (line 1) to bottom (line 25). Each line has 40 (or 80) columns. The WIDTH command allows you to select the number of columns.

The LOCATE statement positions the cursor on the screen. The cursor column and line coordinates are returned by the POS(0) and CSRLIN functions.

Characters are usually displayed, using the PRINT or PRINT USING statements, at the cursor position from left to right on each line, from line 1 to 24. When the cursor passes to line 25, lines 1 to 24 are moved one line up the screen.

Line 25 will usually display the Function Key values (see KEY statement in Chapter 8). To move the cursor to line 25 and display characters, use KEY OFF, then LOCATE and PRINT statements.

Multiple Display Page

Multiple display pages are allowed in Text Mode. Every statement that reads or writes from the screen is actually reading/writing from or to the active page. The visual page is the page that is shown on the screen, and may be different from the active page. This feature allows you to display a page, while writing another. The active and visual pages may be selected by the SCREEN statement.

GRAPHICS

Statements and Functions

The statements and functions available in Text Mode to display text are:

STATEMENTS	FUNCTIONS
CLS COLOR (Text) LOCATE (Text) PRINT PRINT USING SCREEN WIDTH WRITE	CSRLIN POS SCREEN SPC TAB

In Text Mode you can select the character foreground and background colors, and make characters blink.

If color hardware is installed, 16 different colors are available, specified by code:

0 Black	8 Gray
1 Blue	9 Light Blue
2 Green	10 Light Green
3 Cyan	11 Light Cyan
4 Red	12 Light Red
5 Magenta	13 Light Magenta
6 Yellow	14 Light Yellow
7 White	15 High-Intensity White

In a monochrome system you can only use two colors (black and white), but you can also use shades of gray, underline characters, or display high-intensity characters. (See the COLOR (Text) statement in Chapter 8, for details).

GRAPHICS MODES

In each of the three graphics modes you can still display text, but you can also draw complex pictures.

To display text you can use all the statements, commands and functions available in Text Mode, with the exception of the COLOR (Text) and LOCATE (Text) statements. You have to use the COLOR (Graphics) and LOCATE (Graphics) statements instead.

All points of the screen are addressable through their coordinates. A point on the screen is called a 'pixel' (a contraction of "picture element"), and a line of pixels is called a "scanline".

Statements and Functions

The statements and functions you can use in Graphics Mode to display pictures are:

STATEMENT	FUNCTIONS
CIRCLE COLOR (Medium-Resolution Mode) COLOR (High-Resolution Mode) COLOR (Super-Resolution Mode) DRAW GET (Graphics) LINE LOCATE (Graphics) PAINT PRESET PSET PUT (Graphics) SCREEN VIEW WINDOW	PMAP POINT

GRAPHICS

MEDIUM RESOLUTION MODE (SCREEN 1)

In this mode, there are 320 pixels on the horizontal axis and 200 pixels on the vertical axis. These are numbered from left to right and from top to bottom; thus the upper left corner pixel is (0,0) and the lower right corner pixel is (319, 199).

You can display four colors at a time if a color monitor is used, otherwise the four colors will appear as shades of gray.

Drawing Pictures

When you draw pictures on the screen using the graphics statements (PSET, PRESET, LINE, CIRCLE, PAINT or DRAW), you can specify a color number of 0, 1, 2, or 3. This selects the color from the current "palette".

If you do not specify a color number in a graphics statement, the default is the graphics foreground specified by the COLOR statement, or 3 (if no graphics foreground is given).

The COLOR (Medium-Resolution) statement allows you to specify a palette. There are two palettes available, consisting of 4 colors, as shown in the following table:

PALETTE	COLOR NUMBER			
	0	1	2	3
0	<i>background</i>	green	red	yellow
1	<i>background</i>	cyan	magenta	white

Tab. 5-1 Palettes

Where *background* in the first parameter in the COLOR statement; it is specified as a color code and assigned to color number 0. It also specifies the background color for text.

With a monochrome system, the use of memory is identical: the modes differ only in that the two bits of a pixel are interpreted differently by the hardware: 4 shades of gray are displayed.

Displaying Characters

When you display characters in Medium Resolution Mode, the size of the characters is the same as in Text Mode with a 40-column width specified. The character foreground color is set by the *tforeground* parameter in the COLOR statement (that defaults to color number 3). The character background is set by the *background* parameter in the COLOR statement (that defaults to color number 0, i.e. black).

If color is disabled the character foreground will be 1 (white) and the character background 0 (Black).

HIGH RESOLUTION MODE (SCREEN 2)

In this mode, there are 640 pixels on the horizontal axis and 200 pixels on the vertical axis. These are numbered from left to right and top to bottom, thus the upper left pixel is (0,0) and the lower right is (639, 199).

There are only two colors: black (color number 0) and white (color number 1).

Drawing Pictures

When you draw pictures using the graphics statements, you can still specify a color number 0, 1, 2, or 3.

A color of 0 indicates black and a color of 1 white. A color of 2 is treated as 0, and 3 is treated as 1.

If you do not specify a color number, the default is the graphics foreground specified by the COLOR statement, or 1 (if no graphics foreground is given).

GRAPHICS

The COLOR statement allows you to specify the graphics foreground and background colors and, optionally, an XOR operation between the pixels on the screen and the color of the point in memory.

Displaying Characters

The size of the characters is the same as in 80-column Text Mode.

The character foreground color is 1 (white) and the background color is 0 (black).

SUPER RESOLUTION MODE (SCREEN 3)

In this mode, there are 640 pixels on the horizontal axis and 400 pixels on the vertical axis. These are numbered from left to right and top to bottom, thus the upper left corner pixel is (0,0) and the lower right corner pixel is (639, 399).

There are only two colors: black (color number 0) and white (color number 1).

Drawing Pictures

When you draw pictures using the graphics statements, you can still specify a color number of 0, 1, 2, or 3.

A color number of 0 indicates black and a color number of 1 indicates white. A color number of 2 is treated as 0, and a color number of 3 is treated as 1.

If you do not specify a color number, the default is the graphics foreground specified by the COLOR statement, or 1 (if no graphics foreground is given).

The COLOR (Super Resolution) statement allows you to specify the graphics foreground and background colors and, optionally, an XOR operation between the colors of the pixels on the screen and the colors of the points in memory. The COLOR statement also allows you to specify 'inverse video', when you display characters.

Displaying Characters

The size of the characters is the same as in 80-column Text Mode.

The character foreground color is 1 (white) and the character background 0 (black), unless you specify 'inverse video' by the COLOR statement.

WINDOWS AND VIEWPORTS

In the GW-BASIC graphics environment it is possible to define the following types of window:

- Window
- Viewport
- Text Window

A window is a rectangular area with arbitrary dimensions (not limited by the physical units of the screen) in which you can draw lines, graphs or objects; it is defined using the WINDOW statement.

A viewport is a rectangular section of the screen, onto which the associated window is projected, as shown in Figure 5-2.

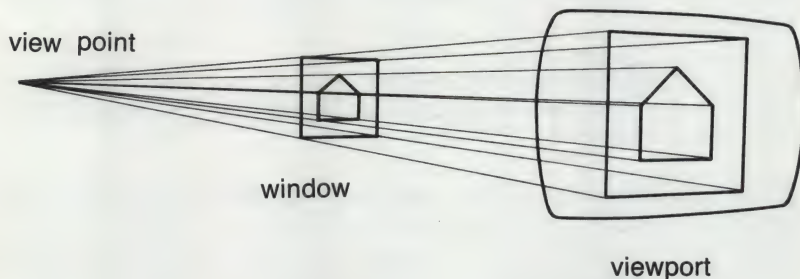


Fig. 5-2 Window and Viewport

GRAPHICS

The command RUN and the statements VIEW and SCREEN, without parameters, define the whole screen as a viewport. The VIEW statement with two pairs of coordinates, defines a subset of the screen as a viewport; the first pair specify the top-lefthand corner of the viewport, and the second pair specify the bottom-right hand corner.

You can define more than one viewport. Each time a VIEW statement is executed, a viewport is defined; this is the "current" viewport. Thus, to change the current viewport, you have to execute another VIEW statement. Only points within the current viewport will be drawn; if points are specified in a graphics statement, that are outside the current viewport, they will not be shown (this is called "clipping").

Apart from defining one or more viewports, you can also define a text window, that is, a rectangular area reserved for text, using the VIEW PRINT and WIDTH statements. You can modify text in the text window (using the Screen Editor), but you cannot perform any graphics functions; the text window also permits automatic scrolling of text.

The VIEW PRINT statement allows you to define a text window by specifying the top and bottom lines of the text; the WIDTH statement allows you to specify the number of columns of text from the left margin of the screen.

TRANSFORMATION USING WINDOW AND VIEW

Varying the area of a window or a viewport changes the relationship between them. Utilizing this feature, you can produce different images from the same design: making it appear smaller or larger (called "zooming"); "clipping" a part of the design; or enlarging a detail (a combination of zooming and clipping).

Whilst varying the area of a window, but maintaining the ratio of the sides, the image produced will not change; if however, the ratio is altered, the image produced will be distorted.

By defining more than one viewport, which partially overlap in a particular way, you can compose a picture on the video made up of parts of other designs (i.e. from different windows).

Another possibility is that of defining different viewports associated with the same window, using only one WINDOW statement and various VIEW statements, all with the SCREEN option specified. This option dictates that the points within the window are projected onto the whole screen, however, only those points within the current viewport are plotted. In this way you can view various parts of the design.

Some examples of these possibilities are given for the WINDOW statement in Chapter 8.

COORDINATES

Each point in a design is specified using coordinates, these are interpreted in the GW-BASIC environment as:

- screen coordinates
- world coordinates

Screen coordinates are positive integer numbers; the origin of the axes is situated in the top-lefthand corner of the screen; the x-axis increases towards the right and the y-axis increases towards the bottom.

World coordinates are signed real numbers; the user defines the position of the origin and the direction in which the y-axis increases.

The WINDOW statement allows you to draw a figure in an area which is not restricted by the bounds of the screen, using a user-defined cartesian coordinate system.

After the execution of a WINDOW statement, the coordinates of all points given in graphics statements are interpreted by GW-BASIC as world coordinates; if a WINDOW statement is not executed the coordinates of all specified points are interpreted as screen coordinates. The DRAW statement is an exception, in that you can only specify screen coordinates and you draw directly in the viewport.

World coordinates are automatically converted by GW-BASIC into screen coordinates for subsequent display within the current viewport.

If the WINDOW statement is executed with no parameters specified, the world coordinates will have a one-to-one correspondence with the screen coordinates.

GRAPHICS

Independently of the definition of world or screen coordinates, the coordinates of a point can be:

- absolute coordinates (the STEP option is not specified in the graphics statement)
- relative coordinates (the STEP option is included in the graphics statement).

Absolute coordinates specify the position of a point with respect to the origin of the coordinate system.

Relative coordinates specify the position of a point with respect to the coordinates of the last referenced point.

The following example illustrates the use of both types of coordinates:

```
10 SCREEN 1 'Medium Resolution
20 PSET (100, 50)
30 PSET STEP (10, -5)
```

Two pixels are drawn; the first at coordinates (100, 50) and the second at coordinates (110,45).

ANIMATION TECHNIQUES

Using the GET statement you can store the graphics image contained within a specified rectangle on the screen into a numeric array. Using the PUT statement you can transfer the image back to the screen in the same position or in another position.

Also, using the PUT statement, you can perform logical operations between the colors of the pixels composing the stored image and the colors of the pixels on the screen.

In order to erase an image from the screen, you only have to put the image back in the same position, using the PUT statement, with the XOR option specified. This is the way in which you animate objects.

In order to make an object appear to be moving, follow these steps:

1. Use the PUT statement to transfer the image (previously stored using the GET statement) onto the screen.
2. Calculate the new position of the rectangle that contains the image.
3. Transfer the image into the same position on the screen, using the PUT statement, with the XOR option specified (in order to erase the image without modifying the background colors).
4. Go back to step 1 specifying, in the PUT statement, the new position of the rectangle that contains the image.

In order to move more than one object, you must repeat the procedure for each object.

6. ADVANCED FEATURES

ABOUT THIS CHAPTER

This chapter is provided for users who want to call machine language subroutines from GW-BASIC, or use event trapping, or Child processes. In particular, it covers:

- Allocation of memory for machine language subroutines
- Loading machine language subroutines into memory
- Calling the subroutine from GW-BASIC and passing parameters to it
- Event Trapping
- Child Processes

This chapter is intended for experienced programmers.

CONTENTS

MACHINE LANGUAGE SUBROUTINES	6-1	USR FUNCTION	6-9
MEMORY ALLOCATION	6-1	EVENT TRAPPING	6-12
LOADING SUBROUTINES INTO MEMORY	6-2	THE ON GOSUB STATEMENT	6-14
USING THE POKE STATEMENT	6-2	THE RETURN STATEMENT	6-14
USING THE BLOAD COMMAND	6-3	GW-BASIC AND CHILD PROCESSES	6-15
CALLING THE SUBROUTINE FROM GW-BASIC	6-4	HARDWARE	6-16
CALL STATEMENT	6-4	THE FILE SYSTEM	6-16
CALLS STATEMENT	6-9	MEMORY MANAGEMENT	6-17

ADVANCED FEATURES

MACHINE LANGUAGE SUBROUTINES

You may call machine language subroutines from your GW-BASIC program with the `USR` function or the `CALL` or `CALLS` statements.

The `USR` function allows you to call a machine language subroutine to return a value in the same way you call GW-BASIC intrinsic functions. However, we recommend that you use the `CALL` or `CALLS` statement for calling machine language programs from GW-BASIC. These statements can pass multiple arguments. In addition, the `CALL` statement is compatible with more versions of BASIC than is the `USR` function.

MEMORY ALLOCATION

Before loading a machine language subroutine into memory you must set aside enough memory space to contain it. There are two ways of doing this.

Using the first method you allocate space for the machine language subroutines in the GW-BASIC Data Segment (DS), using the `CLEAR` statement or the `/M:` option of the `GW BASIC` command. Only the higher addresses of the GW-BASIC Data Segment can be used to hold machine language subroutines.

If more stack space is needed when a machine language subroutine is called, you can save the GW-BASIC stack and set up a new stack for the machine language subroutine. The GW-BASIC stack must be restored, however, before you return from the subroutine.

Using the second method you allocate space for machine language subroutines outside of the GW-BASIC Data Segment, using the `DEF SEG` statement giving the start address of a suitable segment. This address must be greater than the address of the end of the Data Segment (64K plus the start address of the Data Segment).

Which of the two methods you choose is essentially dependent on how much memory is available.

LOADING SUBROUTINES INTO MEMORY

A machine language subroutine can be loaded into memory in several ways, the most simple being to use the BLOAD command. Also, you could SHELL a program that exits, but stays resident. As a third choice, you could execute a program that exits but stays resident, and then run GW-BASIC.

The following guidelines must be observed if you choose to BLOAD, or read and poke, an EXE file into memory:

1. Make sure the subroutines do not contain any long references, address offsets that exceed 64K or that take the user out of the code segment. These long references require handling by the EXE loader.
2. Skip over the first 512 bytes of the linker's output file (EXE), then read in the rest of the file.

The following two sections illustrate two standard ways of loading machine language subroutines.

USING THE POKE STATEMENT

Machine language subroutines can be POKEd into memory as follows: after assembling the subroutine, you should create DATA statements containing the value in hexadecimal of each byte of code (represented as &Hxx). The subroutine is then POKEd into the specified area of memory, byte by byte, in a loop.

The subroutine may then be called using the USR function or the CALL statement. If you use the USR function, then the subroutine entry address must be defined with a DEF USR statement. This defines the USR function call offset into the current segment. If you use the CALL statement, then the subroutine entry address is the value of the numeric variable entered just after CALL. This variable must contain the offset into the current segment. In both cases, the segment is defined by the DEF SEG statement.

ADVANCED FEATURES

USING THE BLOAD COMMAND

A machine language subroutine can be BLOADED into memory as follows:

1. Firstly, create an .EXE file of the subroutine using the linker, then load GW-BASIC under DEBUG by entering:

DEBUG GWBASIC.EXE

2. To determine the location of GW-BASIC in memory, display and record the values contained in registers CS, IP, SS, SP, DS, and ES, using the R command (for use in step 5).
3. Load the .EXE file into high memory using DEBUG, overlaying the transient section of COMMAND.COM.
4. To determine the subroutine memory location, display the registers using the R command. The CS and IP register values should be recorded for use in steps 6 and 7.
5. Reset the register values to their original values as recorded in step 2 using the R command. Breakpoints may optionally be included in the subroutine using the G command which branches to the GW-BASIC entry point.
6. Load your application program. Modify the DEF SEG and either the DEF USR statement or the CALL variable to correspond with the subroutine memory location as defined in step 4 (i.e., the CS register value for DEF SEG and the IP register value for the DEF USR or CALL variable).
7. The subroutine memory area should be BSAVED in GW-BASIC direct mode, using both the CS and IP register values from step 4, and the assembler listing or LINK map code length.
8. Ensure that your application program contains a DEF SEG with the appropriate CS register value, followed by a BLOAD statement.

BLOAD can place a subroutine in an alternate location if the subroutine is self-relocatable. Possible alternatives include an unused screen or file buffer, or a string variable area. (Refer to the BLOAD command and VARPTR Function.) In this case, remember also to modify the associated DEF SEG statement.

9. Finally, the updated application program should be saved onto disk.

Note

If GW-BASIC is run under DEBUG, DEBUG is loaded first, as a precaution against being overwritten. Any breakpoints, or the SYSTEM command, returns control to DEBUG.

CALLING THE SUBROUTINE FROM GW-BASIC

CALL STATEMENT

The CALL statement is the recommended way of calling machine language programs with GW-BASIC. It is preferable to the USR function unless you are running programs that already contain USR functions (for reasons of compatibility).

The syntax of the CALL statement is:

CALL *numvar* [(*variable* [, *variable*] ...)]

Where

numvar contains the offset into the current segment that is the starting point in memory of the subroutine being called

variable is the variable that is passed to the subroutine as an argument

The current segment is either the default (the GW-BASIC Data Segment) or that which has been specified in the most recently executed DEF SEG statement.

ADVANCED FEATURES

Invoking the CALL statement causes the following to occur:

1. For each variable specified in the statement, the two-byte offset of the variable's location within the GW-BASIC Data Segment is pushed onto the stack.
2. The return address specified in the code segment (CS), and offset (IP) are pushed onto the stack.
3. Control is transferred to the machine language routine using the segment address, which is given in the last executed DEF SEG statement and the offset given in *numvar*.

The following diagrams illustrate the state of the stack at the time the CALL statement is executed, and during execution of the called subroutine, respectively.

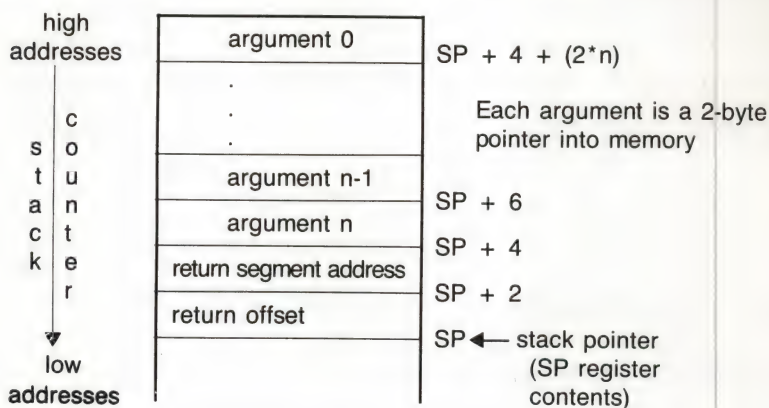


Fig. 6-1 Stack Layout when CALL Statement is Activated

After the CALL statement has been activated, the subroutine has control. Arguments may be referenced by moving the stack pointer (SP) to the base pointer (BP) and adding a positive offset to BP.

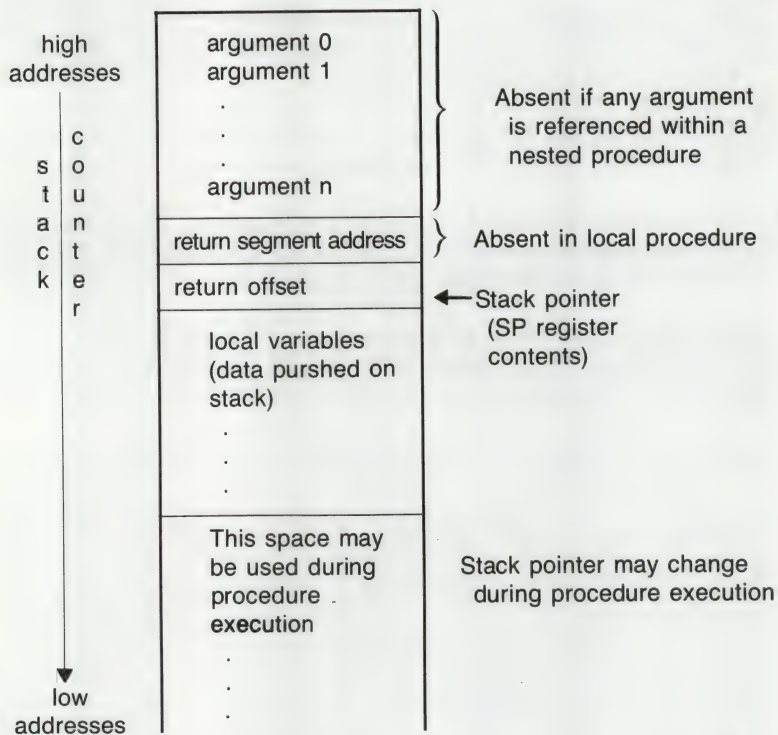


Fig. 6-2 Stack Layout During Execution of a CALL Statement

Observe the following rules when coding a subroutine:

1. The called routine must preserve segment registers DS, ES, SS, and BP. If interrupts are disabled in the routine, they must be enabled before exiting. The stack must be cleaned up on exit.
2. The called subroutine must know the number and length of the arguments passed. The following routine shows an easy way to reference arguments:

```

push    BP
mov     BP,SP
add     BP,(2*number of arguments)+4

```

ADVANCED FEATURES

Then:

argument 0 is at BP
argument 1 is at BP-2
argument n is at BP-2*n

(number of arguments = $n + 1$)

3. Variables may be allocated either in the Code Segment or on the stack. Be careful not to modify the return segment and offset stored on the stack.
4. The called subroutine must clean up the stack. A preferred way to do this is to return to the main program using a `RET n` statement (where n is two times the number of arguments in the argument list) to adjust the stack to the start of the calling sequence.
5. Values are returned to GW-BASIC by including in the argument list the name of the variable that will receive the result.
6. If the argument is a string, the argument's offset points to 3 bytes which, as a unit, are called the "string descriptor". Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

Note

If the argument is a string literal, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, you must add `+"'"` to the string literal in the program. For example, use:

```
20 A$ = "BASIC" + "'"
```

This will force the string literal to be copied into string space. Then the string may be modified without affecting the program.

7. The contents of a string may be altered by user routines, but the descriptor must not be changed. Do not write past the end-of-string. GW-BASIC cannot correctly manipulate strings if their lengths are modified by external routines.

8. Data areas needed by the runtime must be allocated either in the CODE segment of the user routine or on the stack. It is not possible to declare a separate data area in the user machine language routine.

Example

```
100 DEF SEG = &H8000
110 VAR = &H7FA
120 CALL VAR (A,B$,C)
```

```
.
.
.
```

Line 100 sets the segment to 80000 Hex. The value of variable VAR is added into the address as the low word after the DEF SEG value is left shifted 4 bits, i.e. multiplied by 16 (this is a function of the microprocessor, not of GW-BASIC). Here VAR is set to &H7FA, so that the call to VAR will execute the subroutine at location 80000:7FA Hex (absolute address 8007FA Hex).

The following sequence in 8086 assembly language demonstrates access to the arguments passed. The returned result is stored in the variable C.

```
PUSH BP           ;Set up pointer to arguments
MOV BP,SP
ADD BP,(4 + 2*3)
MOV BX,[BP-2]     ;Get address of B$ descriptor.
MOV CL,[BX]       ;Get length of B$ in CL.
MOV DX,1[BX]      ;Get addr of B$ text in DX.
.
.
MOV SI,[BP]       ;Get address of 'A' in SI.
MOV DI[BP-4]      ;Get pointer to 'C' in DI.
MOVS WORD         ;Store variable 'A' in 'C'.
POP BP           ;Restore pointer.
RET 6             ;Restore stack, return.
```


ADVANCED FEATURES

Note

The called subroutine must know the variable type for the numeric arguments. In the previous example, the instruction:

MOVS WORD

will copy only two bytes. This is fine if variables A and C are integer. You would have to copy four bytes if the variables were single precision format and copy 8 bytes if they were double precision.

CALLS STATEMENT

The CALLS statement should be used to access subroutines that were written using MS-FORTRAN calling conventions. CALLS works just like CALL, but the arguments are passed as segmented addresses, rather than as unsegmented addresses.

Because MS-FORTRAN routines need to know the segment value for each argument passed, the segment is pushed and then the offset is also pushed. For each argument, four bytes are pushed rather than 2, as in the CALL statement. Therefore, if your assembler routine is entered using the CALLS statement, *n* in the RET statement is four times the number of arguments.

USR FUNCTION

Although using the CALL statement is the recommended way of calling machine language routines, the USR function is also available for this purpose. This ensures compatibility with programs that contain USR functions.

The format of the USR function is:

USR [*n*] (*argument*)

Where

n is a digit from 0 to 9. It specifies which USR routine is being called. If *n* is omitted, USR0 is assumed.

argument is any numeric or string expression, passed as an argument to the subroutine. The argument must always be specified, even if it isn't required by the subroutine (i.e. a dummy argument).

For each USR function, a corresponding DEF USR statement must be executed to define the offset with respect to the starting address of the current segment. This offset and the currently active DEF SEG address determine the starting address of the subroutine.

When the USR function call is made, register AL contains a value that specifies the type of argument. The value in AL may be one of the following:

2	two-byte integer (two's complement)
3	string
4	single precision floating-point number
8	double precision floating-point number

If the argument is a number, the BX register points to the Floating-Point Accumulator (FAC) where the argument is stored. FAC is an 8-byte area in the GW-BASIC Data Segment.

If the argument is an integer:

FAC-2 contains the upper 8 bits of the argument
FAC-3 contains the lower 8 bits of the argument

ADVANCED FEATURES

If the argument is a single precision floating-point number (24-bit mantissa):

FAC-2 contains the middle 8 bits of mantissa.

FAC-3 contains the lowest 8 bits of mantissa.

If the argument is a double precision floating-point number (56-bit mantissa):

FAC-7 through FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the DX register points to 3 bytes which, as a unit, are called the "string descriptor". Byte 0 of the string descriptor contains the length of the string (0 to 255 characters). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in the GW-BASIC data segment. If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy the program this way.

Usually, the value returned by a USR function is the same type (integer, string, single precision, or double precision) as the argument that was passed to it.

GW-BASIC has extended the USR function interface to allow calls to MAKINT and FRCINT. This allows access to these routines without giving their absolute addresses. The address ES:BP is used as an indirect far pointer to the routines FRCINT and MAKINT.

To call FRCINT using a USR routine use `CALL DWORD ES:[BP]`.
To call MAKINT using a USR routine use `CALL DWORD ES:[BP + 4]`.

Example

```
110 DEF USR0 = &H8000 'Assumes user gave /M:32767
120 X=5
130 Y=USR0(X)
140 PRINT Y
```

The type (numeric or string) of the variable receiving the function call must be consistent with that of the argument passed.

EVENT TRAPPING

Event trapping allows the transfer of control to a program line when a certain event occurs. Control is transferred as if a GOSUB statement had been executed to the trap routine starting at the specified line number. The trap routine, after servicing the event, executes a RETURN statement that causes the program to resume execution at the place where it was when the event trap occurred.

The events that can be trapped are receipt of characters from a communication port (ON COM (n) GOSUB), detection of certain keystrokes (ON KEY (n) GOSUB), time passage (ON TIMER (n) GOSUB), or emptying of the background music queue (ON PLAY (n) GOSUB).

Event trapping is controlled by the following statements:

Syntax 1 (to turn on trapping)

{ COM(n) | KEY(n) | TIMER | PLAY } ON

Syntax 2 (to turn off trapping)

{ COM(n) | KEY(n) | TIMER | PLAY } OFF

Syntax 3 (to temporarily turn off trapping)

{ COM(n) | KEY(n) | TIMER | PLAY } STOP

ADVANCED FEATURES

Remarks

COM(n) where n is the number of the communications port.

Typically, the COM trap routine will read an entire message from the specified port before returning. We do not recommend using the COM trap for single character messages because at high baud rates the overhead of trapping and reading for each character may allow the interrupt buffer for COM to overflow.

KEY(n) where n is a trappable key number. Trappable keys are numbered 1 through 20.

Note that KEY(n) ON is not the same statement as KEY ON. KEY(n) ON sets an event trap for the specified key. KEY ON displays the values of all the function keys on the twenty-fifth line of the screen.

When GW-BASIC is in direct mode function keys maintain their standard meanings.

When a key is trapped, that occurrence of the key is destroyed. Therefore, you cannot subsequently use the INPUT or INKEY\$ statements to find out which key caused the trap. So if you wish to assign different functions to particular keys, you must set up a different subroutine for each key, rather than assigning the various functions within a single subroutine.

TIMER The ON TIMER(n) GOSUB statement (where n is a numeric expression representing a number of seconds since the previous midnight) can be used to perform background tasks at defined intervals.

PLAY The ON PLAY(n) GOSUB statement (where n is a number of notes left in the music buffer) is used to retrieve more notes from the background music queue, to permit continuous background music during program execution.

THE ON GOSUB STATEMENT

The ON GOSUB statement sets up a line number for the specified event trap. The format is:

ON {COM(*n*) | KEY(*n*) | TIMER(*n*) | PLAY(*n*) } GOSUB *linenum*

A *linenum* of zero disables trapping for that event.

When an event is ON and if a non-zero line number has been specified in the ON GOSUB statement, every time GW-BASIC starts a new statement it will check to see if the specified event has occurred (e.g., a COM character has come in). When an event is OFF, no trapping takes place, and the event is not remembered even if it takes place.

When an event is STOPped, no trapping takes place, but the occurrence of that event is remembered so that an immediate trap will take place when the associated event ON statement is executed.

When a trap is made for a particular event, the trap automatically causes a STOP on that event, so recursive traps can never occur. A return from the trap routine automatically executes an ON statement unless an explicit OFF has been performed inside the trap routine.

Note that once an error trap takes place, all trapping is automatically disabled. In addition, event trapping will never occur when GW-BASIC is not executing a program.

THE RETURN STATEMENT

When an event trap is in effect, a GOSUB statement will be executed as soon as the specified event occurs. For example, the statement:

ON KEY(10) GOSUB 1000

specifies that the program go to line 1000 as soon as Function Key F10 is pressed. If a simple RETURN statement is executed at the end of this subroutine, program control will return to the statement following the one where the trap occurred. When the RETURN statement is executed, its corresponding GOSUB return address is cancelled from the stack.

ADVANCED FEATURES

GW-BASIC includes the RETURN *linenum* enhancement, which lets processing resume at a definable line. Normally, the program returns to the statement immediately following the GOSUB statement when the RETURN statement is encountered. However, RETURN *linenum* enables you to specify another line. If not used with care, however, this capability may cause problems. Assume, for example, that your program contains:

```
10 ON KEY(10) GOSUB 1000
20 FOR I = 1 TO 10
30 PRINT I
40 NEXT I
50 REM NEXT PROGRAM LINE

200 REM PROGRAM RESUMES HERE

1000 'FIRST LINE OF SUBROUTINE
.
.
.
1050 RETURN 200
```

If the Function Key F10 is pressed while the FOR/NEXT loop is executing, the subroutine will be performed, but program control will return to line 200 instead of completing the FOR/NEXT loop. The original GOSUB entry will be cancelled by the RETURN statement, and any other GOSUB, WHILE, or FOR, that was active at the time of the trap will remain active. The current FOR context will also remain active, and a "FOR without NEXT" error may result.

GW-BASIC AND CHILD PROCESSES

Through the use of the SHELL command, GW-BASIC is able to use one of the most powerful features of MS-DOS: the ability to create child processes. SHELL enables you to run part of a GW-BASIC program, temporarily exit to MS-DOS to perform a specified function, and return to the GW-BASIC program at the statement after the SHELL command to proceed with the rest of the program.

GW-BASIC will produce a child program when it uses the SHELL command. It is not possible for GW-BASIC to totally protect itself from its children. When a SHELL command is executed, many things may be going on. For example, files may be OPEN and devices may be in use.

The following guidelines will help to prevent child processes from harming the GW-BASIC environment.

HARDWARE

In general, it is recommended that the state of all hardware be preserved during a SHELL command. The implementation interface provides a way for performing this task. However, it may be necessary to request that you refrain from using certain devices within child processes which are executed using the GW-BASIC SHELL command. Specific areas of concern are as follows:

1. Screen Device - Child processes might modify screen mode parameters. However, useful information may be displayed by a child process.
2. Interrupt Vectors - Save and restore interrupt vectors the child intends to use.
3. Other hardware - Many devices are placed in a specific state by GW-BASIC. Such devices may include an Interrupt Controller, Counter Timers, DMA Controller, I/O Latch, and Uarts. These devices may be utilized by the child process without the user being aware of any limitations.

THE FILE SYSTEM

A Child that alters any file open in the GW-BASIC parent may have disastrous results.

If it is necessary to update such files, they should be CLOSED in the parent before doing a SHELL, then re-OPENed upon return to the GW-BASIC parent. (See "Re-direction of Standard Input and Standard Output" in Chapter 8, under the GWBASIC command).

ADVANCED FEATURES

MEMORY MANAGEMENT

1. Before GW-BASIC SHELLs to COMMAND it will try to free any memory it is not using with one exception: when GW-BASIC is run with the /M: switch. In this case, GW-BASIC must assume that you intended to load a routine in the top of GW-BASIC's Memory Block. This prevents GW-BASIC from "compressing the workspace" before doing the SHELL. For this reason SHELL may fail on an "Out of memory" error when using the /M: switch.

The preferred method is to load machine language subroutines before GW-BASIC is run. This can be accomplished by placing "Pocket Code" at the end of machine language subroutines that allows them to exit to MS-DOS and stay resident. For example:

```
CSEG      SEGMENT CODE
          .
          .
          .
          ;Machine language subroutine
          .
          .
          .
          RET      ;Last instruction
START::
          INT      27H ;Terminate, stay resident
CSEG      ENDS
          END      START
```

be sure to "load" these subroutines before GW-BASIC by running them. The AUTOEXEC.BAT file is very useful for this.

2. A Child should never "terminate and stay resident". Doing so may not leave GW-BASIC enough room to expand it's workspace to the original size. If GW-BASIC cannot restore the workspace, all files are closed, the error message "SHELL can't continue" is printed, and GW-BASIC exits to MS-DOS.
3. There is no restriction in the machine independent portion of GW-BASIC which prohibits GW-BASIC from running as a Child of GW-BASIC. However, due to the complications which arise from this configuration, it may not be advisable to use this capability.

7. ASYNCHRONOUS COMMUNICATIONS

ABOUT THIS CHAPTER

This chapter describes how GW-BASIC may be used to support RS232 asynchronous communications with other computers and peripherals.

This chapter is intended for experienced programmers interested in setting up and using asynchronous communications.

CONTENTS

OPENING COMMUNICATIONS FILES	7-1
COMMUNICATION I/O	7-1
COMMUNICATION I/O FUNCTIONS	7-1
THE INPUT\$ FUNCTION FOR COMMUNICATION FILES	7-2
AN EXERCISE IN COMMUNICATION I/O	7-3

ASYNCHRONOUS COMMUNICATIONS

OPENING COMMUNICATIONS FILES

The OPEN COMmunications statement allocates a buffer for input and output in a similar manner as the OPEN statement for disk files. Refer to Chapter 8 for a full description.

COMMUNICATION I/O

Since the communication port is opened as a file, all Input/Output statements that are valid for disk files are valid for COM.

COM sequential input statements are the same as those for disk files. They are: INPUT #, LINE INPUT #, and the INPUT\$ function.

COM sequential output statements are the same as those for disk, and are: PRINT #, PRINT # USING, and WRITE #.

Refer to the INPUT and PRINT statements in Chapter 8 for details of coding syntax and usage.

The GET and PUT statements are only slightly different for COM files. See the "GET (COM Files)" and "PUT (COM Files)" statements described in Chapter 8.

COMMUNICATION I/O FUNCTIONS

The most difficult aspect of asynchronous communication is being able to process characters as fast as they are received. At rates above 2400 bps., it may be necessary to suspend character transmission from the host computer long enough to "catch up". This can be done by sending XOFF (CHR\$(19)) to the host and XON (CHR\$(17)) when ready to resume.

GW-BASIC provides three functions which help in determining when an "over-run" condition is imminent. These are:

LOC(f) Returns the number of characters in the input buffer waiting to be read. The input buffer can hold more than 255 characters (determined by the /C: switch). If there are more than 255 characters in the buffer, LOC(f) returns 255. Since a string is limited to 255 characters, this practical limit means that you do not have to test for string size before reading data into it. If fewer than 255 characters remain in the buffer, LOC(f) returns the actual count.

- LOF(*f*)** Returns the amount of free space in the input buffer. That is, *size*-LOC(*f*), where *size* is the size of the communications buffer as set by the /C: option. LOF may be used to detect when the input buffer is reaching its maximum capacity.
- EOF(*f*)** If true (-1), indicates that the input buffer is empty. Returns false (0) if any characters are waiting to be read.

Possible Errors

"Communication Buffer Overflow"

If a read is attempted after the input buffer is full, (i.e. LOF(*f*) returns 0).

"Device I/O Error"

If any of the following line conditions are detected on reception: Over-run Error (OE), Framing Error (FE), or Break Interrupt (BI). The error is reset by subsequent inputs but the character causing the error is lost.

"Device Fault"

If Data Set Ready (DSR) is lost during I/O.

THE INPUT\$ FUNCTION FOR COMMUNICATION FILES

The INPUT\$ function is preferable to the INPUT # and LINE INPUT # statements when reading COM files, since all ASCII characters may be significant in communications. INPUT # is least desirable because input stops when a comma (,) or CR is received and LINE INPUT # terminates when a CR is received.

INPUT\$ allows all characters read to be assigned to a string. Remember from the coding rules that INPUT\$ (*n*,*f*) will return *n* characters from the #*f* file. The following statements are therefore the most efficient for reading a COM file:

```
10 WHILE NOT EOF(1)
20 A$ = INPUT$ (LOC(1), # 1)
30 ...
40 Process data returned in A$
50 ...
60 WEND
```

ASYNCHRONOUS COMMUNICATIONS

The above statements return the characters in the buffer into A\$ and process them, provided there are characters in the buffer. If there are more than 255 characters, only 255 will be returned at a time to prevent "String Overflow". If this is the case, EOF(1) is false and input continues until the input buffer is empty. The sequence of events is therefore simple, concise, and fast.

AN EXERCISE IN COMMUNICATION I/O

The following program enables your Personal Computer to be used as a conventional terminal. Besides Full Duplex communication with another computer, the TTY program allows received data to be "Downloaded" to a file. Conversely, a disk file may be "Up-loaded" (transmitted) to another machine.

In addition to demonstrating the elements of Asynchronous communication, this program should be useful in transferring GW-BASIC programs and Data to and from your system.

```
10 REM
20 REM *** RS232 test program ***
30 REM
40 SCREEN 0,0
50 KEY OFF:CLS:CLOSE
60 DEFINT A-Z
62 LOCATE 25,1
64 PRINT STRING$(60," ")
70 FALSE=0:TRUE=NOT FALSE
80 MENU=5
90 XOFF$=CHR$(19):XON$=CHR$(17)
92 LOCATE 25,1:PRINT "TTY Async Program"
94 LOCATE 1,1:LINE INPUT "Baud?"; SPEED$
96 LOCATE 25,1:PRINT"TTY Async Program"
100 ON COM(1) GOSUB 730
110 COMFIL$="COM1:" + SPEED$ + ",E,7"
120 OPEN COMFIL$ AS 1
```

Fig. 7-1 The TTY Program


```

130 REM
140 REM *** talk mode ***
150 REM
160 CLS
170 LOCATE 25,1 : PRINT "RS232 test program running in TALK
    MODE";
180 PAUSE = FALSE
190 LOCATE 1,1
200 A$ = INKEY$: IF A$ = "" THEN 220
210 IF ASC(A$) = MENU THEN 290 ELSE PRINT #1,A$;
220 IF EOF(1) THEN 200
230 IF LOC(1) > 50 THEN PAUSE = TRUE: PRINT #1,XOFF$;
240 A$ = INPUT$(LOC(1),1)
250 PRINT A$;:IF LOC(1) > 0 THEN 230
260 IF PAUSE THEN PAUSE = FALSE:PRINT #1,XON$;
270 GOTO 200
280 REM
290 REM *** command mode ***
300 REM
310 CLS
320 LOCATE 25,1:PRINT "RS232 test program running in
    COMMAND MODE";
330 LOCATE 1,1
340 INPUT "FILE";DSKFIL$
350 LOCATE 1,1:PRINT STRING$(80," "):LOCATE 1,1
360 INPUT "(T)ransmit or (R)eceive";TXRX$
370 IF TXRX$ = "T" THEN OPEN DSKFIL$ FOR INPUT AS
    3:GOTO 580
380 IF TXRX$ = "R" THEN 410
390 GOTO 350
400 REM
410 REM *** file receive mode ***
420 REM
430 LOCATE 25,32 : PRINT "FILE RECEIVE MODE";
440 OPEN DSKFIL$ FOR OUTPUT AS 3
450 IF EOF(1) THEN GOSUB 520
460 IF LOC(1) > 50 THEN PAUSE = TRUE:PRINT #1,XOFF$
470 A$ = INPUT$(LOC(1),1)
480 PRINT #3,A$;
490 IF LOC(1) > 0 THEN 460

```

Fig 7-1 The TTY Program (cont.)

ASYNCHRONOUS COMMUNICATIONS

```
500 IF PAUSE THEN PAUSE = FALSE:PRINT #1,XON$;
510 GOTO 450
520 FOR I=1 TO 5000
530 IF NOT EOF(1) THEN RETURN
540 NEXT I
550 CLOSE # 3
560 RETURN 140
570 REM
580 REM *** file transmit mode ***
590 REM
600 LOCATE 25,32 : PRINT"FILE TRANSMIT MODE";
610 COM(1) ON
620 XFLAG = 1
630 WHILE NOT EOF(3)
640 A$=INPUT$(1,3)
650 WHILE XFLAG=0 :WEND
660 PRINT #1,(A$);
670 WEND
680 COM(1) OFF
690 PRINT #1,CHR$(26)
700 CLOSE 3
710 GOTO 140
720 REM
730 REM *** XON/XOFF receiving routine ***
740 REM
750 IF EOF(1) THEN RETURN
760 B$=INPUT$(LOC(1),1)
770 IF LEN(B$)=2 THEN 790
780 IF B$=XOFF$ THEN 810
790 XFLAG = 1
800 RETURN
810 XFLAG=0
820 RETURN
```

Fig. 7-1 The TTY Program (cont.)

Notes on the TTY Programming Example

Line No.	Comments
10-90	Define the screen attributes and initialize program variables.
92-96	Permits the user to enter the communication speed (bps), from the keyboard.
100	Specifies the line number of the first statement of the COM trap routine associated with channel number 1.
110-120	Open and initialize communications channel 1 with the speed entered from the keyboard (bps), parity even, and 7 data bits.
170	Displays a message indicating the operation mode (Talk Mode) on the 25th screen line.
180-260	Send characters entered from Keyboard to channel number 1, and displays characters received from the channel. Statement 210 transfer control to statement 290 (where Command Mode is entered), if the user enters CTRL E .
320	Displays a message indicating the new mode (Command Mode) on the 25th screen line.
340	Asks the user to enter the name of the file to transmit or receive, depending on the character (T or R) entered upon execution of statement 360.
370	If the user enters T, open the specified file for INPUT and branches to statement 580.
380	If the user enters R, branches to statement 410 (where Receive Mode is entered).

ASYNCHRONOUS COMMUNICATIONS

- 410-560 The program is in Receive Mode, as displayed by statement 430 on the 25th screen line. Statement 440 opens the specified file for OUTPUT. Statement 450 checks if characters are pending on the receive buffer. If no characters are pending, control is transferred to statement 520, otherwise to the following statement (line 460).
- 460 Checks if the number of characters in the receive buffer is greater than 50. If the number is greater than 50, it sends an XOFF\$ character to the channel to stop transmission.
- 470-480 If the number of characters in the receive buffer is less than or equal to 50, characters are read from the receive buffer and written to the file.
- 490-510 Check if there are still characters on the receive buffer. If yes, control is transferred to statement 460. If not an XON\$ character is sent (if an XOFF\$ was sent before) and control is transferred to statement 450.
- 520-540 A FOR/NEXT loop is activated to wait until characters arrive in the receive buffer. If no character arrives within the specified number of iterations, the Receive Mode is exited. Control is then transferred to statement 550 where the file is closed, and then to statement 140 returning in "TALK MODE". If characters arrive control is transferred to statement 460.

570-710

The program is in Transmit Mode, as displayed by statement 600 on the 25th Screen Line ("File Transmit Mode"). The file has already been opened for input at statement 370. Statement 610 enables COM trapping. Statements 630 to 670 form a WHILE/WEND loop to read and transmit the file (statement 640 reads one character at a time and statement 660 sends it to the communications channel). The character transmission is suspended if an XOFF\$ character is received (see statement 650). The character transmission is resumed if an XON\$ character is received. Statements 680 to 710 disable COM trapping, send an EOF character, close the file and return to "Talk Mode".

730-820

Form the COM trap routine. Statement 750 checks if characters are pending in the receive buffer. If no character is pending a RETURN is executed. If two characters are pending, the transmission of characters is enabled (statement 790) and the routine is exited (RETURN). Two characters in the receive buffer means that both an XON\$ and an XOFF\$ have been received. If only one character is pending the transmission of characters is disabled (if this character is XOFF\$) or enabled (if this character is XON\$).

PART II

8. COMMANDS, STATEMENTS AND FUNCTIONS

ABOUT THIS CHAPTER

This chapter gives a full description of each command, statement and function available under GW-BASIC, with examples for use.

CONTENTS

INTRODUCTION	8-1	CALLS Statement	8-36
COMMANDS	8-2	CDBL Function	8-37
STATEMENTS	8-5	CHAIN Statement	8-38
NON-I/O STATEMENTS	8-5	CHDIR Command	8-43
I/O STATEMENTS	8-11	CHR\$ Function	8-45
FUNCTIONS	8-18	CINT Function	8-46
NUMERIC FUNCTIONS (returning a numeric value)	8-18	CIRCLE Statement	8-47
		CLEAR Command	8-53
STRING FUNCTIONS (returning a string value)	8-23	CLOSE Statement	8-56
ABS Function	8-26	CLS Statement	8-57
ASC Function	8-27	COLOR Statement (Text Mode)	8-59
ATN Function	8-28	COLOR Statement (Medium-resolution Graphics)	8-62
AUTO Command	8-29	COLOR Statement (High-resolution Graphics)	8-66
BEEP Statement	8-30	COLOR Statement (Super-resolution Graphics)	8-69
BLOAD Command	8-31		
BSAVE Command	8-33		
CALL Statement	8-35		

COM(<i>n</i>) Statement	8-71	EOF Function	8-111
COMMON Statement	8-73	ERASE Statement	8-112
CONT Command	8-77	ERDEV and ERDEV\$ Functions	8-113
COS Function	8-78	ERR and ERL Functions	8-115
CSNG Function	8-79	ERROR Statement	8-117
CSRLIN Function	8-80	EXP Function	8-119
CVI, CVS, CVD Functions	8-80	FIELD Statement	8-120
DATA Statement	8-82	FILES Command	8-123
DATE\$ Function and Statement	8-84	FIX Function	8-125
DEF FN Statement	8-86	FOR...NEXT Statements	8-126
DEF SEG Statement	8-88	FRE Function	8-130
DEF USR Statement	8-90	GET (COM files) Statement	8-131
DEFINT / SNG / DBL / STR Statements	8-92	GET (Files) Statement	8-132
DELETE Command	8-93	GET (Graphics) Statement	8-133
DIM Statement	8-94	GOSUB...RETURN Statements	8-136
DRAW Statement	8-99	GOTO Statement	8-138
EDIT Command	8-105	GWBASIC Command	8-139
END Statement	8-106	HEX\$ Function	8-146
ENVIRON Statement	8-107	IF...GOTO...ELSE and IF...THEN... ELSE Statements	8-147
ENVIRON\$ Function	8-109		

INKEY\$ Function	8-150	LOCATE (Text) Statement	8-191
INP Function	8-152	LOCATE (Graphics) Statement	8-195
INPUT Statement	8-153	LOCK Statement	8-199
INPUT # Statement	8-156	LOF Function	8-201
INPUT\$ Function	8-157	LOG Function	8-202
INSTR Function	8-160	LPOS Function	8-203
INT Function	8-162	LPRINT and LPRINT USING Statement	8-204
IOCTL Statement	8-162	LSET and RSET Statements	8-205
IOCTL\$ Function	8-165	MERGE Command	8-207
KEY Statement	8-166	MID\$ Function and Statement	8-208
KEY(n) Statement	8-171	MKDIR Command	8-211
KILL Command	8-174	MKIS, MKS\$, MKD\$ Functions	8-213
LCOPY Command	8-175	NAME Command	8-215
LEFT\$ Function	8-176	NEW Command	8-217
LEN Function	8-177	OCT\$ Function	8-217
LET Statement	8-178	ON COM(n) GOSUB Statement	8-218
LINE Statement	8-179	ON ERROR GOTO Statement	8-221
LINE INPUT Statement	8-182	ON...GOSUB and ON...GOTO Statements	8-224
LINE INPUT # Statement	8-184		
LIST Command	8-185		
LLIST Command	8-187		
LOAD Command	8-188		
LOC Function	8-190		

ON KEY(<i>n</i>) GOSUB Statement	8-225	PSET Statement	8-281
ON PLAY(<i>n</i>) GOSUB Statement	8-229	PUT (COM files) Statement	8-283
ON TIMER (<i>n</i>) GOSUB Statement	8-232	PUT (Files) Statement	8-284
OPEN Statement	8-234	PUT (Graphics) Statement	8-286
OPEN COM Statement	8-242	RANDOMIZE Statement	8-290
OPTION BASE Statement	8-246	READ Statement	8-291
OUT Statement	8-247	REM Statement	8-294
PAINT Statement	8-248	RENUM Command	8-296
PEEK Function	8-253	RESET Command	8-297
PLAY Statement	8-254	RESTORE Statement	8-298
PLAY(<i>n</i>) Function	8-258	RESUME Statement	8-299
PLAY {ON OFF STOP} Statement	8-259	RIGHT\$ Function	8-300
PMAP Function	8-260	RMDIR Command	8-302
POINT Function	8-263	RND Function	8-304
POKE Statement	8-266	RUN Command	8-305
POS Function	8-267	SAVE Command	8-307
PRESET Statement	8-268	SCREEN Function	8-309
PRINT Statement	8-269	SCREEN Statement	8-310
PRINT USING Statement	8-272	SGN Function	8-315
PRINT # and PRINT # USING Statements	8-279	SHELL Command	8-316
		SIN Function	8-318

SOUND Statement	8-319	VIEW PRINT Statement	8-349
SPACE\$ Function	8-322	WAIT Statement	8-350
SPC Function	8-323	WHILE...WEND Statements	8-351
SQR Function	8-324	WIDTH Statement	8-353
STOP Statement	8-325	WINDOW Statement	8-358
STR\$ Function	8-326	WRITE Statement	8-368
STRING\$ Function	8-327	WRITE # Statement	8-369
SWAP Statement	8-329		
SYSTEM Command	8-330		
TAB Function	8-331		
TAN Function	8-332		
TIME\$ Statement and Function	8-333		
TIMER Function	8-336		
TIMER Statement	8-336		
TRON/TROFF Commands	8-337		
UNLOCK Statement	8-339		
USR Function	8-341		
VAL Function	8-342		
VARPTR Function	8-343		
VARPTR\$ Function	8-345		
VIEW Statement	8-346		

INTRODUCTION

This chapter contains information on all the commands, statements and functions used in GW-BASIC. An introduction to each constitutes the first few pages, followed by a complete alphabetical list. For each command, statement or function, this chapter will give the purpose, the syntax, the characteristics, one or more examples, and possibly remarks.

Statements and Commands

It is sometimes difficult to distinguish a GW-BASIC statement from a GW-BASIC command, as both may be used in a program or an immediate line; however:

- GW-BASIC statements are generally used in program lines and entered in sequence to form a program.
- GW-BASIC commands are generally used in immediate lines to manipulate programs and for utility purposes, such as listing programs or clearing the memory.

Functions

A function may be thought of as an algorithm returning a single value. There are both user functions and built-in functions. It can be called simply by stating its name, followed (in parentheses) by one or more arguments representing the values that the function parameters are to assume. A function may be called both from a program and an immediate line (with the exception of EOF that may only be called from a program line).

A function may be numeric if it returns a numeric value, or a string function if it returns a string value.

All built-in functions are listed in this chapter. For information on how to calculate mathematical functions which may be easily derived from built-in functions, refer to Appendix B.

COMMANDS

The following is a list of all the commands used in GW-BASIC.

AUTO

Generates a line number after every carriage return.

BLOAD

Loads a memory image file into memory.

BSAVE

Saves sections of the main memory on the specified file.

CHDIR

Changes the current directory.

CLEAR

Clears all numeric variables to zero, all string variables to null, and closes all open files. Options set the highest memory location available for use by GW-BASIC, and the amount of stack space.

CONT

Resumes program execution after a **CTRL BREAK** has been typed or a **STOP** or **END** statement has been executed.

DELETE

Erases program lines.

COMMANDS, STATEMENTS AND FUNCTIONS

EDIT

Lets you change a specified program line.

FILES

Displays the names of files residing on the specified directory.

GW BASIC

Initializes GW-BASIC and the operating environment (GW BASIC is an MS-DOS command, not a GW-BASIC command).

KILL

Deletes a disk file.

LCOPY

Dumps the screen (text and graphics) to the line printer.

LIST

Lists the current program to the screen or a specified file or device.

LLIST

Lists the current program on the printer.

LOAD

Loads a program into memory from a specified drive and, optionally runs it.

MERGE

Merges the current program with a specified file previously saved in ASCII format.

MKDIR

Permits the creation of a new directory on a specified disk.

NAME

Changes the name of a disk file.

NEW

Deletes the current program and clears all variables, allowing you to enter a new program.

RENUM

Changes the line numbers of the current program.

RESET

Closes all open data files on all drives.

RMDIR

Removes a directory from a specified disk.

RUN

Runs the current program or loads a program from disk and runs it.

COMMANDS, STATEMENTS AND FUNCTIONS

SAVE

Saves the current program on disk and gives it a name.

SHELL

Loads and executes another program (.EXE, .COM or .BAT).

SYSTEM

Closes all open data files and returns control to MS-DOS.

TRON

Causes the line number of each statement executed to be listed.

TROFF

Stops the line number listing initiated by TRON.

STATEMENTS

This part lists all the GW-BASIC statements. They are divided into two categories: I/O (Input/Output) Statements and Non-I/O Statements.

NON-I/O STATEMENTS

CALL[S]

Transfers control to a machine language subroutine. The CALLS statement is executed in the same way as the CALL statement and should be used when accessing routines written with FORTRAN calling conventions.

CHAIN

Transfers control and passes variables to another program.

COM(*n*)

COM (*n*) ON enables, COM (*n*) OFF disables, and COM (*n*) STOP suspends event trapping of communications activity on the specified channel.

COMMON

Defines a common area which is not erased by the CHAINED program, and allows you to pass variables from one program to another.

DATE\$

Sets the current date.

DEF FN

Defines and names a user-written function.

DEF SEG

Assigns the current "segment" address.

DEF USR

Enables access to a machine language subroutine by specifying the starting address.

DEFINT, DEFSNG, DEFDBL and DEFSTR

Declare the variable type in accordance with the letter(s) specified.

COMMANDS, STATEMENTS AND FUNCTIONS

DIM

Specifies the array name, the number of dimensions and the subscript upper bound per dimension. The DIM statement may specify one or more arrays.

END

Terminates program execution, closes all open data files, and returns to command level.

ENVIRON

Allows modification of parameters in GW-BASIC's Environment String Table.

ERASE

Releases space and variable names previously reserved for arrays.

ERROR

Simulates the occurrence of a GW-BASIC error, or generates a user defined error.

FOR/NEXT

Allow a series of statements to be performed in a loop a given number of times.

GOSUB/RETURN

GOSUB transfers control to a GW-BASIC subroutine by branching to the specified line. RETURN transfers control to the statement following the most recent GOSUB (or ON...GOSUB) executed.

GOTO

Transfers control to a specified program line.

IF ... GOTO ... ELSE

or

IF ... THEN ... ELSE

Make a decision regarding program flow based on the result of a specified condition.

KEY

Sets a function key to automatically type any sequence of characters. Other options allow you to enable or disable the function key display from the 25th line, or to list the function key values.

KEY(n)

KEY(n) ON enables, KEY(n) OFF disables, and KEY(n) STOP suspends event trapping of the specified key.

LET

Assigns a value to a variable.

MID\$

Replaces a part of a string with another string.

ON COM(n) GOSUB

Specifies the first line number of a subroutine to be activated as soon as characters arrive in the communications buffer.

COMMANDS, STATEMENTS AND FUNCTIONS

ON ERROR GOTO

Enables error trapping and specifies the first line number of a subroutine to be executed if an error occurs.

ON ... GOSUB

Calls one of several specified subroutines, depending on the value of a specified expression.

ON ... GOTO

Branches to one of several specified line numbers, depending on the value of a specified expression.

ON KEY(*n*) GOSUB

Specifies the first line number of a subroutine to be executed when a specified function key or cursor control key is pressed.

ON PLAY(*n*) GOSUB

Specifies the first line number of a subroutine to be executed when the music buffer contains fewer than *n* notes.

ON TIMER(*n*) GOSUB

Causes an event trap every *n* seconds.

OPTION BASE

Defines the minimum value for array subscripts.

PLAY

Plays music in accordance with a string which specifies the notes to be played, and the way in which the notes are to be played.

PLAY { ON | OFF | STOP }

PLAY ON enables, PLAY OFF disables, and PLAY STOP suspends play event trapping.

POKE

Writes a byte into a memory location.

RANDOMIZE

Reseeds the random number generator.

REM

Allows explanatory remarks to be inserted in a program.

RESTORE

Permits DATA statements to be re-read either from the beginning of the internal file or from a specified line.

RESUME

Continues program execution after an error trapping routine has been performed.

STOP

Terminates program execution then returns to command level.

COMMANDS, STATEMENTS AND FUNCTIONS

SWAP

Exchanges the values of two variables.

TIMES

Sets the current time.

TIMER

TIMER ON enables, TIMER OFF disables, and TIMER STOP suspends real time event trapping.

WAIT

Suspends program execution while monitoring the status of a machine input port.

WHILE/WEND

Loop through a series of statements as long as a given condition remains true.

I/O STATEMENTS

BEEP

Activates the bell.

CIRCLE

Draws a circle (or an ellipse) with the specified center and radius (Graphics mode only).

CLOSE

Terminates I/O to a file or device.

CLS

Erases all or part of the screen.

COLOR (Text Mode)

Sets the text foreground and background.

COLOR (Medium Resolution Mode)

Defines the palette background and foreground colors. In addition, the default graphics foreground and background colors, and the text foreground color can be defined.

COLOR (High Resolution Mode)

Defines the (default) graphics foreground, the graphics background and the text foreground colors.

COLOR (Super Resolution Mode)

Defines the (default) graphics foreground, background and text foreground colors.

DATA

Creates an "internal" file, i.e. a sequence of data belonging to the program. Each data item will then be assigned to a program variable by a READ statement.

COMMANDS, STATEMENTS AND FUNCTIONS

DRAW

Draws an object as specified by the contents of a string expression.
(Graphics Mode only.)

FIELD

Allocates space for variables in a random file buffer.

GET (COM Files)

Reads a specified number of bytes into the communications buffer.

GET (Files)

Reads a record from a random disk file into a random buffer.

GET (Graphics)

Reads points from a screen area.

INPUT

Allows input from the keyboard during program execution.

INPUT #

Reads data items from a sequential disk file and assigns them to program variables.

IOCTL

Sends a "Control Data" string to a Character Device Driver anytime after the Driver has been OPENed.

LINE

Draws either a line or a rectangle, or a filled rectangle (Graphics Mode only).

LINE INPUT

Inputs an entire line (up to 254 characters) to a string variable, without the use of delimiters.

LINE INPUT #

Reads an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

LOCATE (Text)

Moves the cursor to the specified position on the active page. LOCATE may also turn the cursor on and off and define the size of either the user cursor, or both the user and overwrite cursors.

LOCATE (Graphics)

Moves the cursor to the specified position. LOCATE may also turn the cursor on and off and define the shape and blinkrate of the cursor.

LPRINT

Prints data on the printer.

LPRINT USING

Prints data to the printer using a specified format.

LSET/RSET

Moves data from memory to a random file buffer.

COMMANDS, STATEMENTS AND FUNCTIONS

OPEN

Allows I/O to a file or device.

OPEN COM

Opens and initializes a communications channel for input/output.

OUT

Transmits a byte to an output port.

PAINT

Paints an enclosed area on the screen with a specified color (Graphics Mode only).

PRESET

Draws a point at the specified position on the screen (Graphics Mode only).

PRINT

Outputs data to the screen.

PRINT USING

Outputs data to the screen using a specified format.

PRINT #

Writes data sequentially to a disk file.

PRINT # USING

Writes data sequentially to a disk file, using a specified format.

PSET

Illuminates a pixel at a specified position on the screen (Graphics mode only).

PUT (COM Files)

Writes a specified number of bytes to a communications file.

PUT (Files)

Writes a record from a random buffer to a random disk file.

PUT (Graphics)

Transfers the graphics image stored in an array to the screen.

READ

Reads values from one or more DATA statements and assigns them to variables.

RESET

Closes all open data files on all drives.

SCREEN

Sets the specifications for the display screen.

COMMANDS, STATEMENTS AND FUNCTIONS

SOUND

Produces sound via a speaker.

VIEW

Defines subsets of the screen called 'viewports', into these, window contents will be mapped. (Graphics Mode only.)

VIEW PRINT

Sets the boundary of the text window.

WIDTH

Sets the line width in characters.

WINDOW

Defines the logical dimensions of the current viewport. (Graphics Mode only).

WRITE

Writes data to the screen.

WRITE #

Writes data to a sequential file.

FUNCTIONS

This section lists all built-in (or intrinsic) functions used in GW-BASIC. They are divided into two categories: numeric and string functions. Further classes are defined within each category.

NUMERIC FUNCTIONS (returning a numeric value)

Arithmetic

ABS

Returns the absolute value of a numeric expression.

ATN

Returns the arctangent of the argument.

CDBL

Converts a given numeric expression to a double precision number.

CINT

Converts any numeric argument to an integer by rounding the fractional portion.

COS

Returns the cosine of the argument.

CSNG

Converts any numeric argument to a single precision number.

COMMANDS, STATEMENTS AND FUNCTIONS

EXP

Returns e (base of natural logarithms) to the power of the argument.

FIX

Returns the truncated integer part of the argument.

INT

Returns the largest integer that is equal to, or less than the argument.

LOG

Returns the natural logarithm of a positive argument.

RND

Returns a random number between 0 and 1.

SGN

Returns 1 if the argument is positive, 0 if the argument is zero, and -1 if the argument is negative.

SIN

Calculates the sine of the argument.

SQR

Returns the square root of a positive numeric expression.

TAN

Returns the tangent of the argument.

String-Related

ASC

Returns a numeric value that is the ASCII code for the first character of a given string.

{ **CVI** | **CVS** | **CVD** }

Convert string values to numeric values.

INSTR

Searches for the first occurrence of a given substring in a string, and returns the position at which the match is found.

LEN

Returns the number of characters in a given string.

VAL

Converts the string representation of a number to its numeric value.

I/O and Miscellaneous

CSRLIN

Returns the current line (row) position of the cursor.

COMMANDS, STATEMENTS AND FUNCTIONS

EOF

Indicates that the end of a file has been reached.

ERDEV

Holds the actual value of a Device error.

ERR

Returns the error code.

ERL

Returns the number of the line which contains the error.

FRE

Returns the number of bytes in memory not being used by GW-BASIC.

INP

Returns the byte read from a port.

LOC

Returns the current position in the file.

LOF

Returns the number of bytes allocated to the file.

LPOS

Returns the current position of the print head within the printer buffer.

PEEK

Returns the byte read from the specified memory location.

PLAY

Returns the number of notes currently in the music background buffer.

PMAP

Maps physical coordinates to world coordinates or world coordinates to physical coordinates (Graphics Mode only).

POINT

Returns the color of a pixel on the screen or the current graphics coordinate (Graphics Mode only).

POS

Returns the current cursor column position.

SCREEN

Returns the ASCII code (0-255) or the color number for the character at the specified screen location.

TIMER

Returns a single-precision number indicating the seconds that have elapsed since midnight or system reset.

COMMANDS, STATEMENTS AND FUNCTIONS

USR

Calls a machine language subroutine.

VARPTR

VARPTR (*variable*) returns the address of *variable*.
VARPTR (*filenum*) returns the starting address of the disk I/O buffer (for sequential files) or the starting address of the FIELD buffer (for random files).

STRING FUNCTIONS (returning a string value)

General

CHR\$

Returns a one-character string whose ASCII code is the value of the argument.

LEFT\$

Returns a substring extracting a number of characters to the left of a given string, as specified by the *length* parameter.

MID\$

Returns a substring from a specified string.

RIGHT\$

Returns a substring from a specified string, extracting its rightmost characters.

SPACES\$

Returns a string of a specified number of spaces.

STRING\$

Returns a string of specified length whose characters all have the same ASCII code or equal the first character of a given string.

I/O and Miscellaneous

DATE\$

Retrieves the current date.

ENVIRON\$

Allows you to retrieve the specified Environment String from GW-BASIC's Environment String Table.

ERDEV\$

Holds the name of the device causing the error if it was a character device.

HEX\$

Returns a string which represents the hexadecimal value of the decimal argument.

INKEY\$

Returns a one- or two-character string read from the keyboard or a null string if no character is pending at the keyboard.

COMMANDS, STATEMENTS AND FUNCTIONS

INPUT\$

Returns a string of characters read from the keyboard or from a file.

IOCTL\$

Returns a "Control Data" string from a Character Device Driver that is OPEN.

{MKI\$ | MKS\$ | MKD\$}

Change numeric values to string type values.

OCT\$

Returns a string which represents the octal value of the decimal argument.

SPC

Skips *n* spaces in a PRINT, LPRINT, or PRINT # statement.

STR\$

Returns the string representation of the value of a specified numeric expression.

TAB

Tabs the cursor or the print head to a specified position, in PRINT, LPRINT, or PRINT # statements.

TIME\$

Retrieves the current time.

VARPTR\$

Returns a character form of the memory address of the variable.

ABS Function

Returns the absolute value of a numeric expression.

ABS(*numexp*)

Characteristics

The returned value will always be positive or zero.

Example

```
Ok
PRINT ABS(8*(-6))
48
Ok
```

Example

```
DISTANCE = ABS(START-FINISH)
```

Example

```
IF ABS(DELTA) <= LIMIT THEN STOP
```

ASC Function

Returns the ASCII decimal code for the first character of a given string.

ASC(*stringexp*)

Characteristics

The ASC function returns the ASCII code (0-255) corresponding to the first character of the string *stringexp* . See Appendix C for a complete list of all ASCII codes.

If *stringexp* is null, an "Illegal function call" error is returned.

See the CHR\$ function, for ASCII-to-string conversion. CHR\$ is the inverse of the ASC function.

Example

The following example shows that the ASCII code for capital letter 'T' is 84.

```
10 X$ = "TEST"  
20 PRINT ASC(X$)  
RUN  
84  
Ok
```


ATN Function

Returns the arctangent of the argument.

ATN(*numexp*)

Characteristics

The evaluation of ATN is performed in single precision, unless /D is supplied in the GWBASIC command line.

The result is expressed in radians and falls in the range $-\pi/2$ to $\pi/2$ (where $\pi = 3.141593$).

Example

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
  1.249046
Ok
```

Example

```
100 IF ATN(N) < PI/2.0 THEN PRINT "ANGLE 90 DEGREES"
```

AUTO Command

Generates a line number after every carriage return. AUTO is only used in immediate mode.

AUTO [*linenum*][, [*increment*]]

Where

SYNTAX ELEMENT	MEANING
<i>linenum</i>	Is the line number used to commence numbering lines.
<i>increment</i>	Is the value added to a line number to produce the next line number.

Characteristics

AUTO begins numbering at *linenum* and increments each subsequent line number by *increment*. The default for both values is 10. If *linenum* is followed by a comma but *increment* is not specified, the last increment specified in an AUTO command is assumed. If no preceding AUTO command was given, an increment of 10 is assumed.

If AUTO generates a line number that is already being used, an asterisk is displayed after the number to warn the user that any input will overwrite the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing **CTRL BREAK** (or **CTRL C**); the current line is not saved and GW-BASIC returns to command level.

Examples

AUTO

Generates line numbers 10, 20, 30, 40

AUTO 100,20

Generates line numbers 100, 120, 140

AUTO 200,

Generates line numbers 200, 220, 240, 260, ...

The increment is 20 because 20 was the increment in the last AUTO command.

AUTO,15

Generates line numbers 0,15,30,45, ...



BEEP Statement

Activates the bell.

BEEP

Characteristics

PRINT CHR\$(7); will send an ASCII BEL character, which will have the same effect.

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
130 IF X < 20 THEN BEEP
```

BLOAD Command

Loads a memory image file into memory.

BLOAD {*filespec*|*pathname*} [, *offset*]

Where

SYNTAX ELEMENT	MEANING
<i>filespec</i> (or <i>pathname</i>)	Is a string expression that specifies the file to be loaded. If the device is omitted, the MS-DOS default drive is assumed.
<i>offset</i>	Is an integer expression in the range 0 to 65535. This is the offset into the segment declared by the last DEF SEG statement at which loading is to start.

Characteristics

The BLOAD and BSAVE statements allow you to load into memory, and save on a file, machine language routines. When these routines are resident in memory, they can be CALLED from your GW-BASIC program by a CALL statement.

The BLOAD and BSAVE statements also allow you to load and save any portion of memory, for instance you can save and display screen images (specifying the screen buffer as the current segment by a DEF SEG statement).

If *offset* is omitted, the offset specified at BSAVE is assumed, and the file is loaded into the same location from which it was saved.

If *offset* is specified, a DEF SEG statement should be executed before the BLOAD. When *offset* is given, GW-BASIC assumes the user wants to BLOAD at an address other than the one saved. The last known DEF SEG address will be used. If no DEF SEG statement has been given, the GW-BASIC data segment will be used as the default (because it is the default for DEF SEG).

Warning

BLOAD does not perform an address range check. It is therefore possible to load a file anywhere in memory. You must be careful not to load over GW-BASIC, or the operating system.

Example

```
10 DEF SEG 'Restore segment to GW-BASIC's DS
20 BLOAD "B:PROG1", &HF000 'Load at offset F000
```

Example

```
10 'Load the screen buffer
20 DEF SEG = &HB800 'Point segment at screen buffer
30 BLOAD "FILE1",0 'Load FILE1 into screen buffer
```

COMMANDS, STATEMENTS AND FUNCTIONS

Note the DEF SEG statement in 20 and the offset of 0 in 30: this guarantees that the correct address is used.

An example under BSAVE illustrates how FILE1 was saved.

BSAVE Command

Saves sections of the main memory on the specified file.

BSAVE {*filespec* | *pathname* } , *offset* , *length*

Where

SYNTAX ELEMENT	MEANING
<i>filespec</i> (or <i>pathname</i>)	Is a string expression which specifies the name of the file to be saved, and optionally a device. If the device is omitted, the MS-DOS default drive is assumed.
<i>offset</i>	Is an integer expression in the range 0 to 65535. This is the offset into the segment declared by the last DEF SEG.
<i>length</i>	Is an integer expression in the range 1 to 65535, specifying the length of the memory image to be saved.

Characteristics

A memory image file is a byte-for-byte copy of what is in memory.

The BLOAD and BSAVE statements allow you to load into memory, and save on a file, machine language routines. When these routines are resident in memory, they can be CALLED from your GW-BASIC program by a CALL statement.

The BLOAD and BSAVE statements also allow you to load and save any portion of memory, for instance you can save and display screen images (specifying the screen buffer as the current segment by a DEF SEG statement).

A DEF SEG statement should be executed before the BSAVE. The last known DEF SEG address is always used for the save.

Example

```
10 'Save PROG1
20 DEF SEG = &H6000
30 BSAVE "PROG1",&HF000,256
```

This examples saves 256 bytes starting at 60000:F000 in the file PROG1.

Example

```
10 'Save the screen buffer
20 DEF SEG = &HB800 'Point segment at screen buffer
30 BSAVE "A:FILE1",0,16384 'Save screen buffer in FILE1
```

The DEF SEG statement must be used to set up the Segment address to the screen buffer. The offset of 0 and the length 16384 specify that the entire 16K screen buffer is to be saved.

CALL Statement

Transfers control to a machine language subroutine. CALL is usually used in a program.

CALL *numvar* [(*variable* [, *variable*]...)]

Where

SYNTAX ELEMENT	MEANING
<i>numvar</i>	Is a numeric variable. It must equate to the starting address of the machine language routine. The address is an offset into the current memory segment as set by the last DEF SEG statement.
<i>variable</i>	Is a numeric or string variable which serves as an argument to pass data between the main program and the machine language routine.

Characteristics

The CALL statement is one way to transfer program flow to an external subroutine. You can transfer control to an external subroutine also by use of the USR function.

Example

```
110 MYROUT = &HD000
120 CALL MYROUT (I,J,K)
.
.
.
```

CALLS Statement

The CALLS statement is the same as the CALL statement with the exceptions given below under "Characteristics".

CALLS *numvar* [(*variable* [, *variable*]...)]

Where

SYNTAX ELEMENT	MEANING
<i>numvar</i>	Is a numeric variable. It contains the address that is the starting point in memory of the subroutine being CALLED
<i>variable</i>	Is a numeric or string variable which has to be passed as an argument to the machine language subroutine.

Characteristics

The CALLS statement is similar to CALL, except that the segmented addresses of all arguments are passed (CALL passes unsegmented addresses). CALLS should be used when accessing routines written with FORTRAN calling conventions, since all FORTRAN parameters are call-by-reference segmented addresses.

CALLS uses the segment address defined by the most recently executed DEF SEG statement to locate the routine being called.

CDBL Function



Converts a given numeric expression to a double precision number.

CDBL(*numexp*)

Example

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
    454.67 454.6700134277344
    Ok
```

CHAIN Statement

Transfers control and passes variables to another program. CHAIN is only used in a program.

```
CHAIN [ MERGE ] { filespec | pathname } [ , [ linenum ] [ , [ ALL ]  
[ , DELETE range ] ] ]
```

Where

SYNTAX ELEMENT	MEANING
<i>filespec</i> (or <i>pathname</i>)	Is a string expression which specifies the name of the called program file and optionally the drive. If the drive is omitted the MS-DOS default drive is assumed.
<i>linenum</i>	Is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line. The parameter is not affected by a RENUM command.
<i>range</i>	Is the range of line numbers to be deleted. These line numbers are affected by the RENUM command.

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

IF...	THEN...
the MERGE option is used	<p>a MERGE operation is performed with the current program and the CHAINED program. The CHAINED program must be an ASCII file. If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (MERGEing may be thought of as "inserting" the program lines on disk into the program in memory). The MERGE option leaves the files open, preserves the current OPTION BASE setting, and preserves variable types and user-defined functions, for use by the CHAINED program.</p> <p>User-defined functions should be placed before any CHAIN MERGE statements in the program. Otherwise, the user-defined functions will be undefined after the merge is complete.</p>
the MERGE option is omitted	<p>the CHAINing program is lost (except common variables) before loading the CHAINED program. CHAIN does not preserve variable types or user functions. Thus, any DEFtype or DEF FN statements containing shared variables must be repeated in the CHAINED program.</p>
the ALL option is used	<p>every variable in the current program is passed to the CHAINED program</p>

IF...	THEN...
<p>the ALL option is used and <i>linenum</i> is omitted</p>	<p>two commas must be inserted between the <i>filespec</i> (or pathname) and the ALL option. For example:</p> <pre>100 CHAIN "NEXTPROG",,ALL</pre> <p>is correct, but:</p> <pre>100 CHAIN "NEXTPROG",ALL</pre> <p>is incorrect. In this case, GW-BASIC assumes that ALL is a variable name and evaluates it as a line number.</p>
<p>the ALL option is omitted</p>	<p>the current program must contain one or more COMMON statements to list the variables that are passed.</p>
<p>the DELETE option is used</p>	<p>a section of the current program will be deleted before loading the CHAINED program.</p> <p>DELETE is often used with MERGE and 'line' options, to load overlays. After an overlay is brought in, it is usually desirable to delete it so a new overlay may be brought in.</p>

Remarks

Before running a CHAINED program, CHAIN carries out a RESTORE. This resets the pointer to the beginning of the internal data file.

COMMANDS, STATEMENTS AND FUNCTIONS

Example 1

CHAIN is used in different ways in two programs below. In the first the two string arrays are dimensioned, and declared, as common variables. When the first program gets to line 80, it chains to the second program (PROG 2), which loads the two elements of the B\$ array. At line 80 of PROG 2, control chains back to the first program, beginning execution at line 90. This process can be observed through the descriptive text that prints as the programs execute.

```
10 REM THIS PROGRAM DEMONSTRATES CHAINING USING
COMMON TO PASS VARIABLES
20 REM SAVE THIS MODULE ON DISK AS "PROG1"
  USING THE A OPTION.
30 DIM A$(2),B$(2)
40 COMMON A$(),B$()
50 A$(1) = "VARIABLES IN COMMON MUST BE ASSIGNED"
60 A$(2) = "VALUES BEFORE CHAINING."
70 B$(1) = "" : B$(2) = ""
80 CHAIN "PROG2"
90 PRINT: PRINT B$(1): PRINT: PRINT B$(2): PRINT
100 END
```

```
10 REM THE STATEMENT "DIM A$(2), B$(2)"
  MAY ONLY BE EXECUTED ONCE.
20 REM HENCE, IT DOES NOT APPEAR IN THIS
  MODULE
30 REM SAVE THIS MODULE ON THE DISK AS "PROG2"
  USING THE A OPTION.
40 COMMON A$(), B$()
50 PRINT: PRINT A$(1); A$(2)
60 B$(1) = "NOTE HOW THE OPTION OF SPECIFYING
  A STARTING LINE NUMBER"
70 B$(2) = "WHEN CHAINING AVOIDS THE DIMENSION
  STATEMENT IN 'PROG1'."
80 CHAIN "PROG1", 90
90 END
```

Example 2

In the following example, the MERGE, ALL, and DELETE options are illustrated. After A\$ is loaded in the first program, control chains to line 1010 of the second. At the second program's line 1040, it chains to line 1010 of the third program, keeping all variables and deleting all the second program's lines. Control passes to the third program. This process can be observed through the descriptive text that prints as the programs execute.

```
10 REM THIS PROGRAM DEMONSTRATES CHAINING USING
   THE MERGE, ALL, AND DELETE OPTIONS
20 REM SAVE THIS MODULE ON THE DISK AS 'MAINPRG'
30 A$ = "MAINPRG"
40 CHAIN MERGE "OVLAY1", 1010, ALL
50 END
```

```
1000 REM SAVE THIS MODULE ON THE DISK AS "OVLAY1"
     USING THE A OPTION.
1010 PRINT A$; "HAS CHAINED TO OVLAY1."
1020 A$ = "OVLAY1"
1030 B$ = "OVLAY2"
1040 CHAIN MERGE "OVLAY2", 1010, ALL, DELETE 1000-1050
1050 END
```

```
1000 REM SAVE THIS MODULE ON THE DISK AS "OVLAY2"
     USING THE A OPTION.
1010 PRINT A$; "HAS CHAINED TO"; B$; "."
1020 END
```

CHDIR Command

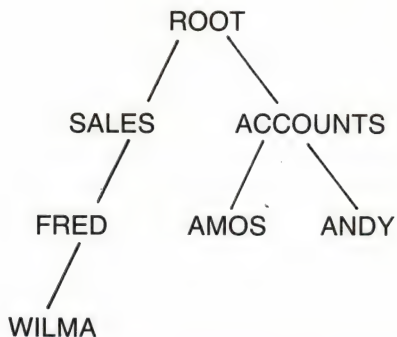
Changes the current directory.

CHDIR *pathname*

Where

SYNTAX ELEMENT	MEANING
<i>pathname</i>	Is a string expression identifying the new directory which is to be the current directory

Examples



Assuming that the diskette on drive B has the directory structure illustrated above, let us change the current directory from ROOT to ACCOUNTS:

CHDIR "B: ACCOUNTS"

ACCOUNTS is now the current directory on drive B.

To change the current directory from ACCOUNTS to ANDY use:

CHDIR "ANDY"

Remarks

Avoid nesting directories too deeply as a result of using MKDIR and CHDIR repeatedly.

Possible Errors

"Bad file name"

"Path not found"

"Path/File Access error"

CHR\$ Function

Returns a one-character string whose ASCII decimal code is the value of the argument.

CHR\$(*n*)

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is an integer expression which must be in the range 0 to 255. It represents an ASCII code. If it is outside the specified range, an "Illegal Function Call" is returned.

Characteristics

CHR\$ is normally used to send a special character to an output device. For instance, the BEL character (CHR\$(7)) could be sent as a preface to an error message, or a form feed (CHR\$(12)) could be sent to clear a terminal screen and return the cursor to the home position.

Examples

```
100 PRINT CHR$(7) 'BEEP
150 PRINT CHR$(LINEFEED%)
200 IF CHR$(INP(IN.PORT%)) = "A" THEN GOSUB 100
```

CINT Function

Converts any numeric argument to an integer by rounding the fractional portion.

CINT(*numexp*)

Characteristics

If *numexp* is not in the range -32768 to 32767, an "Overflow" error occurs.

If the fractional portion of *numexp* is $\geq .5$ the integer part is rounded up; otherwise a truncation occurs.

See the CDBL and CSNG functions for details on converting numbers to the double precision and single precision data type, respectively. See also the FIX and INT functions, both of which return integers.

Examples

```
Ok
PRINT CINT(45.67)
46
Ok
PRINT CINT(-3.71)
-4
Ok
```

CIRCLE Statement

Draws a circle (or an ellipse) with the specified center and radius (Graphics mode only).

CIRCLE [STEP] (*x* , *y*), *radius* [, *color* [, *start* , *end* [, *aspect*]]]

Where

SYNTAX ELEMENT	MEANING
<i>x,y</i>	Are numeric expressions, specifying the coordinates of the center of the circle (or ellipse). They may be absolute coordinates, or relative coordinates (if STEP is included).
<i>radius</i>	Is a numeric expression returning a positive integer value giving the value of the radius of the circle, or the major axis of the ellipse. It is measured in the horizontal direction if <i>aspect</i> < 1, and in vertical direction if <i>aspect</i> > 1. The value of the radius is interpreted as pixels if the WINDOW statement has not been executed, otherwise it is interpreted as screen coordinates.

SYNTAX ELEMENT	MEANING
<i>color</i>	<p>Is an integer expression in the range 0 to 3. It is the color number of the circumference of the circle (or ellipse).</p> <p>In Medium Resolution <i>color</i> chooses the color from the currently selected palette. In High and Super Resolution, 0 or 2 indicates black, and 1 or 3 indicates white.</p> <p>If <i>color</i> is omitted, the color specified for the <i>gforeground</i> parameter in the COLOR statement is assumed; if this parameter has also been omitted, color 3 in Medium Resolution and color 1 in High and Super Resolutions, are assumed.</p>
<i>start,end</i>	<p>Are numeric expressions specifying angles in radians. The range is from $-2 \cdot \pi$ to $2 \cdot \pi$, where $\pi = 3.141593$. They specify where the drawing of the arc of the circle (or ellipse) will begin and end.</p>
<i>aspect</i>	<p>Is a numeric expression returning a positive real value. Due to the difference between the spacing of pixels on the x-and y-axis of the screen, you must specify a value of <i>aspect</i> to draw a true circle with different monitors. The default value of 'aspect' is 5/6 in medium and super resolution and 5/12 in high resolution. This value produces a circle with the standard monitor.</p>

COMMANDS, STATEMENTS AND FUNCTIONS

Drawing Circles and Ellipses

The CIRCLE statement draws circles if you do not specify the *aspect* parameter, and ellipses if you specify a value of *aspect* different from the default value (5/6 in Medium and Super Resolution, and 5/12 in High Resolution).

The *aspect* may be thought of as a fraction, which specifies the number of pixels necessary in order to have equal lengths along the two axes. The numerator specifies the number of pixels along the x-axis and the denominator specifies the number of pixels along the y-axis.

If *aspect* is less than one, then *radius* is measured in pixels in the horizontal direction i.e., it is the x-radius. In this case GW-BASIC draws ellipses with the same width, and varies the height as the parameter varies.

If *aspect* is greater than one, the y-radius is given, and GW-BASIC draws ellipses with the same height and varies the width, as the parameter varies.

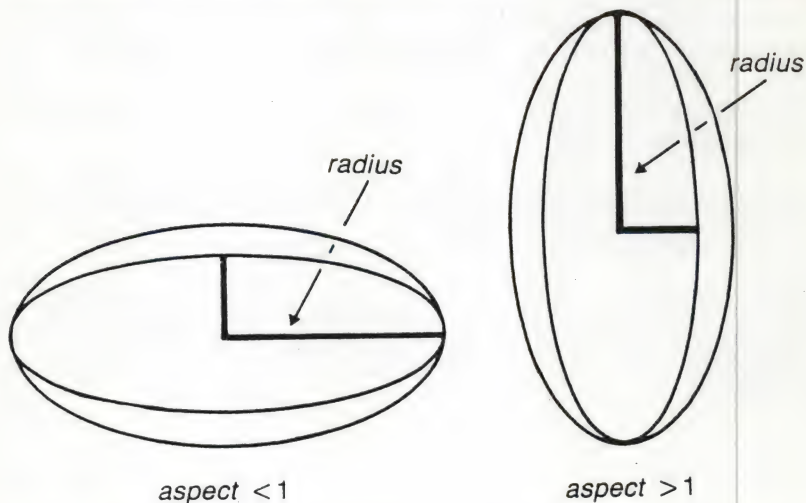


Fig. 8-1 The *aspect* Parameter

It is not possible to transform a circle (drawn with the CIRCLE statement) into an ellipse using the WINDOW and VIEW statements (i.e. by varying the ratio of the sides of the window); it is possible to achieve this if the circle has been drawn (using the LINE statement) as a regular polygon with a large number of sides.

Example

```
10 SCREEN 1
```

```
...
```

```
100 CIRCLE (100,150),50,,,5/18
```

will draw a horizontal ellipse with an x-radius of 50 pixels.

Drawing Arcs

The CIRCLE statement can simply draw part of a circle (or ellipse) i.e. an "arc".

To draw an arc you must enter the *start* and *end* parameters. They specify the first and the second arc endpoint in radians.

The angles are measured from the x-axis in a counterclockwise direction, as shown in Figure 8-2.

For example, the following statement specifies just a quarter of a circle:

```
10 CIRCLE (100,150),50,1,0,3.141593/2
```

The angles must be measured in radians. If you have the angles in degrees, you must convert them to radians before executing the CIRCLE statement. To convert from degrees to radians, multiply by 0.0174532.

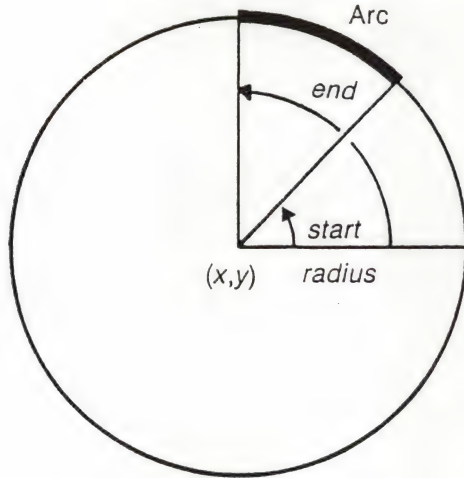


Fig. 8-2 Arc of a Circle

Drawing Rays

The CIRCLE statement can draw a ray from the center of the arc to either arc endpoint.

A negative endpoint generates a ray to that endpoint. The endpoint -0 is not treated as a negative endpoint. To circumvent this limitation, use a small negative number (e.g. -0.001 instead of -0). When both endpoints are negative, both rays are drawn. The minus sign does not affect the arc itself, i.e. the angles will be treated as if they were positive. Note that this is different from adding $2 * \text{Pi}$ (where Pi is 3.141593). The start angle may be greater or less than the end angle. For example:

```
10 SCREEN 1
```

```
...
```

```
100 CIRCLE (100,150),50,1,-0.001,-3.141593/2
```

will draw a quarter of a circle delimited by two rays.

Last Point Referenced

The last point referenced after a circle (or ellipse) has been drawn is the center of the circle (or ellipse).

STEP Option

Coordinates can be absolute, or relative (if the STEP option is specified). Relative coordinates are interpreted as relative to the last referenced point.

For example, if the last point referenced was 100,50, then:

both:

```
CIRCLE (200,200),50
```

and:

```
CIRCLE STEP (100,150),50
```

will draw a circle at 200,200 with radius 50. The first example uses absolute coordinates; the second uses relative coordinates.

Example

The following example draws three intersecting circles and colors the area of intersection.

```
5 SCREEN 1
10 COLOR 0,0,3,0
20 CLS
30 CIRCLE (100,120),90
40 CIRCLE (150,130),120
50 CIRCLE (250,120),100
60 PAINT (180,120)
```

CLEAR Command

Clears all numeric variables to zero, all string variables to null, closes all open files and cancels the effect of any DEF or DIM statements. Options set the maximum memory available for use by GW-BASIC, and the amount of stack space.

CLEAR [, [*memory*][, *stack*]]

Where

SYNTAX ELEMENT	MEANING
<i>memory</i>	Is an integer expression representing a memory location which, if specified, sets the top of memory available to GW-BASIC (i.e. the maximum extension of the GW-BASIC Data Segment). The maximum value allowed is 65536. The default value (i.e. when the parameter is omitted or equals 0) is the maximum current value as specified in the GW-BASIC command (or 65536 if the GWBASIC command does not specify a value).
<i>stack</i>	Is an integer expression whose value sets aside stack space for GW-BASIC. The default is 128 bytes or one-eighth of the available memory, whichever is smaller.

Characteristics

The *memory* parameter should be specified to reserve sufficient space in memory for machine language routines. The *stack* parameter is used to increase the stack space; the amount of space reserved may be critical when a program has several nested GOSUB statements, several nested FOR...NEXT loops and/or PAINT statements which paint complex figures.

If there is not enough space for the program, or for the stack, you will get an "Out of Memory" error.

GW-BASIC allocates string space dynamically. An "Out of string space" error occurs only if there is no free memory left for GW-BASIC to use.

The CLEAR statement does not modify the program in memory but performs the following actions:

- Closes all files
- Clears all COMMON variables
- Resets the stack and frees the space reserved for strings
- Resets all simple numeric variables and numeric array elements to zero
- Resets all simple string variables and string array elements to null
- Releases all disk buffers
- Resets all DEF FN, DEFINT/SNG/DBL/STR, DEF SEG and DEF USR statements

Examples

CLEAR

CLEAR ,32768

CLEAR ,,2000

CLEAR ,32768,2000

CLOSE Statement

Terminates I/O to a file or device. CLOSE is usually used in a program.

CLOSE [[#] *filenum* [, [#] *filenum*]...]

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is the number under which the file was OPENed. A CLOSE with no arguments closes all open files.

Characteristics

The association between a particular file and file number terminates upon execution of a CLOSE statement. The file may then be reOPENed using the same or a different file number; likewise, a file number may now be reused to OPEN any file.

A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always close all disk files automatically (STOP does not close disk files).

COMMANDS, STATEMENTS AND FUNCTIONS

Example

To read the data in a sequential file open for output or append, you must first CLOSE the file and then re-OPEN it in the "I" mode.

```
100 CLOSE #1
110 OPEN "I", #1, "DATA"
```

CLS Statement

Erases all of the screen or a window.

CLS [*n*]

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is an integer expression in the range 0 to 2.

Characteristics

CLS without a parameter clears the entire screen to the current text background color (unless a graphics viewport has been defined) and resets the function key line (if the function key display is enabled). If a viewport has been defined, the current viewport only, will be cleared to the graphics background color. Outputting a formfeed character by typing `CTRL L`, or entering `PRINT CHR$ (12)`, will have the same effect as CLS without parameters. If there is a text window, and no graphics viewport, then CLS will clear only the text window.

CLS 0 clears the entire screen, resetting the function key display.

CLS 1 clears the graphics viewport to the graphics background color (in one of the graphics modes). If no viewport has been defined, this will have no effect.

CLS 2 clears the text window to the text background color, without resetting the function key display.

Remarks

CLS not only erases all or part of the screen, but also returns the cursor to the upper left-hand corner of the screen (in Text Mode).

If you are in Graphics Mode, CLS makes the "last referenced point" the center of the screen.

The screen can also be cleared by pressing `CTRL HOME`, or by modifying the screen mode using the `SCREEN` statement, or the width using the `WIDTH` statement.

Examples

```
10 CLS
```

Clears the screen (or the current viewport, or the text window - as described above)

```
60 CLS 0
```

Clears whole screen

COMMANDS, STATEMENTS AND FUNCTIONS

90 CLS 1

Clears the graphics viewport to graphics background color

110 CLS 2

Clears the text window to text background color

COLOR Statement (Text Mode)

Sets the text foreground and background colors (Text Mode only).

COLOR [*foreground*] [, *background*] [, *dummy*]

Where

SYNTAX ELEMENT	MEANING
<i>foreground</i>	Is a numeric expression rounded to the nearest integer. It must be in the range 0 to 31. Values greater than 15 are interpreted modulo 16. It selects the character foreground color.
<i>background</i>	Is a numeric expression rounded to the nearest integer. It must be in the range 0 to 15, but it is interpreted modulo 8, thus only values from 0 to 7 are taken into consideration. It selects the character background color.

SYNTAX ELEMENT	MEANING
<i>dummy</i>	Is a parameter allowed for compatibility with other BASICs. It will have no effect. It may specify screen border color on other systems.

Characteristics (Color Text Mode)

If you are using a color system the following colors are available for *foreground*:

0 Black	8 Gray
1 Blue	9 Light Blue
2 Green	10 Light Green
3 Cyan	11 Light Cyan
4 Red	12 Light Red
5 Magenta	13 Light Magenta
6 Yellow	14 Light Yellow
7 White	15 High-intensity White

To make characters blink for a specific color, you should set *foreground* equal to 16 plus the color number.

Only colors 0 through 7 are allowed for *background* .

Characteristics (B/W Text Mode)

In a monochrome system the following values can be used for *foreground*.

- 0 Black
- 1 Underlined character with white foreground
- 2-6 Shades of gray
- 7 White

COMMANDS, STATEMENTS AND FUNCTIONS

Adding 8 to the number of the desired color gives you the color in high-intensity.

To make character blink, add 16 to the number of the desired color.

The following values are allowed for *background* :

- 0 - 1 Black
- 2 - 6 Shades of gray
- 7 White

Remarks

Foreground and background colors may be equal. In this case any character displayed is invisible. Changing the foreground or background color will make subsequent characters visible again.

Any parameter may be omitted. Omitted parameters assume the old value.

Upon initialization, the default values are:

- *foreground* = 7 (White)
- *background* = 0 (Black)

That is, if no COLOR statement exists in your program, the system assumes:

COLOR 7,0

Examples

100 COLOR 0,2

This sets a black foreground on a green background with a color screen and a black foreground on a gray background, with a black and white screen.

150 COLOR 15,1

This sets a high-intensity white on a blue background with a color screen, and a high-intensity white on a black background with a black and white screen.

Possible Errors

- If the COLOR statement ends in a comma (,), a "Missing operand" error is returned. For example:

COLOR 2,

is invalid.

- Any parameters outside the specified ranges will result in an "Illegal function call" error. In this case, previous values are retained.

COLOR Statement (Medium-resolution Graphics)

Defines the palette background and foreground colors. In addition, the default graphics foreground and background colors, and the text foreground color can be defined.

COLOR [*background*][, [*palette*][, [*gforeground*] [, [*gbackground*] [, *tforeground*]]]]

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING												
<i>background</i>	<p>Is a numeric expression rounded to the nearest integer. It must be in the range 0 to 31. Values greater than 15 are taken modulo 15. It represents the color code for the character background. This is also the actual color for color number 0 that may be specified in a graphics statement. This parameter also specifies the foreground intensity (the intensity of pixels with values 1, 2 or 3). If the parameter is > 15, high intensity is selected for foreground colors. It defaults to 0 (black) if unspecified.</p>												
<i>palette</i>	<p>Is a numeric expression rounded to the nearest integer. It must be in the range 0 through 255. This selects one of two palettes. A graphics statement can specify a foreground color through a color number (0, 1, 2 or 3) which selects the desired color from the choice provided by the active palette. If the color number is 0, the color specified by <i>background</i> will be selected.</p> <table><tr><td>Palette</td><td>Color 1</td><td>Color 2</td><td>Color 3</td></tr><tr><td>0</td><td>Green</td><td>Red</td><td>Yellow</td></tr><tr><td>1</td><td>Cyan</td><td>Magenta</td><td>White</td></tr></table> <p>Palette 0 is selected when <i>palette</i> is an even number, whereas palette 1 is selected if <i>palette</i> is an odd number. Palette 1 is the default.</p>	Palette	Color 1	Color 2	Color 3	0	Green	Red	Yellow	1	Cyan	Magenta	White
Palette	Color 1	Color 2	Color 3										
0	Green	Red	Yellow										
1	Cyan	Magenta	White										

SYNTAX ELEMENT	MEANING
<i>gforeground</i>	<p>Is a numeric expression rounded to the nearest integer. It must be in the range 0 to 7. Values greater than 3 are interpreted modulo 4. This specifies the graphics foreground which is the default color number when no color parameter is specified in a graphics statement. If <i>gforeground</i> is omitted, 3 is assumed. The graphics foreground is always set to the default value (3) when the SCREEN statement selects new screen mode 1.</p> <p>Any value greater than 3 will cause the bits for each pixel to be XOR'd with screen memory.</p>
<i>gbackground</i>	<p>Is a numeric expression rounded to the nearest integer. It must be in the range 0 to 3. This specifies the graphics background; i.e., the color used when a graphics viewport is cleared with CLS 1, or with just CLS. The default value is 0, which is always selected when a new screen mode is selected. Note that the CLS 1 statement will only clear a viewport if it has been explicitly defined with a VIEW statement.</p>
<i>tforeground</i>	<p>Is a numeric expression rounded to the nearest integer. It must be in the range 0 to 7. Values greater than 3 are interpreted modulo 4. It selects the character foreground color. It defaults to 3. Any value greater than 3 will cause the character to be XOR'd with screen memory.</p>

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

When you enter a CIRCLE, DRAW, LINE, PAINT, PRESET, or PSET statement, you can specify a color number of 0, 1, 2, or 3. This parameter selects the color from the current "palette" as defined by the COLOR statement.

If you do not specify a color number, the default is the graphics foreground (i.e. the value of *gforeground* , or 3 if *gforeground* has not been specified).

When you display text the character foreground will be set by *tforeground* (that defaults to color number 3), and the character background will be set by *background* (that defaults to 0, i.e. black).

Any parameter may be omitted in the COLOR statement. Omitted parameters assume the old value.

Upon initialization the default values are:

- *background* = 0 (black)
- *palette* = 1 (cyan, magenta, white)
- *gforeground* = 3 (white)
- *gbackground* = 0 (black)
- *tforeground* = 3 (white)

That is, if no COLOR statement exists in your program, the system assumes:

COLOR 0, 1, 3, 0, 3

Examples

```
10 SCREEN 1,0  
20 COLOR 10,1,2,0
```

Sets the character background to light green, selects palette 1 (Cyan, Magenta, White), sets the graphics foreground to magenta, and graphics background to light green.

```
100 COLOR ,0
```

The character background stays light green and palette 0 (green, red, yellow) is selected.



COLOR Statement (High-resolution Graphics)

Defines the (default) graphics foreground and the graphics background colors. It also specifies if the text is normal, reverse video or XOR'd with screen memory.

COLOR [*gforeground*][, [*gbackground*] [, [*tforeground*]]

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
<i>gforeground</i>	<p>Is a numeric expression rounded to the nearest integer. It must be in the range 0 to 3. This specifies the graphics foreground which is the default color number for graphics statements. If <i>gforeground</i> is omitted, 1 (white) is assumed.</p> <p>Values of 0 (black) or 1 (white) represent actual colors; 2 and 3 specify an XOR attribute.</p>
<i>gbackground</i>	<p>Is a numeric expression rounded to the nearest integer, whose value may be 0 (black) or 1 (white). This specifies the background color used for clearing graphics viewports. This defaults to 0 (black).</p>
<i>tforeground</i>	<p>Is a numeric expression, rounded to the nearest integer. A zero value result in reverse video text (characters written with black pixels on a white background). A value equal to 1 gives normal text (characters written with white pixels on a black background); this is the default value. Values greater than 1 result in the color of each pixel being XOR'd with the colors of the pixels on the screen.</p>

Characteristics

When you enter a CIRCLE, DRAW, LINE, PAINT, PRESET, or PSET statement in your program, you can specify a color number of 0, 1, 2, or 3. In High Resolution a color number of 0 indicates black and a color number of 1 white. A color number of 2 will be treated as 0, and a color number of 3 will be treated as 1.

If you do not specify a color number, the default is the graphics foreground (i.e. the value of *gforeground* or 1 if *gforeground* has not been specified).

When you display text without specifying that *tforeground* equals 0 (reverse video), the character foreground will be 1 (white) and the background 0 (black).

You can also specify an XOR operation between the pixels on the screen and the pixels in memory, by specifying *tforeground* with a value greater than 1.

Any parameter in the COLOR statement may be omitted. Omitted parameters assume the old values. Upon initialization default values are:

- *gforeground* = 1 (White)
- *gbackground* = 0 (Black)
- *tforeground* = 0 (no XOR attribute for text)

That is, if no COLOR statement exists in your program, the system assumes:

```
COLOR 1,0,0
```

Example

```
SCREEN 2  
COLOR 0,1,0
```

This selects a default black graphics foreground on a white background and no XOR attribute for text.

COLOR Statement (Super-resolution Graphics)

Defines the (default) graphics foreground and graphics background colors. It also specifies if the text is normal, reverse video or XOR'd with screen memory.

COLOR [*gforeground*][, [*gbackground*][, *tforeground*]]

Where

SYNTAX ELEMENT	MEANING
<i>gforeground</i>	Is the same as for high-resolution.
<i>gbackground</i>	Is the same as for high-resolution.
<i>tforeground</i>	Is a numeric expression rounded to the nearest integer. A value of 0 results in reverse video text (characters written with black pixels on a white background). A value of 1 results in normal text (characters written with white pixels on a black background); this is the default value. A value greater than 1 results in the color of each pixel being XOR'd with the color of the pixels on the screen.

Characteristics

When you enter a **CIRCLE**, **DRAW**, **LINE**, **PAINT**, **PRESET**, or **PSET** statement in your program, you can specify a color number of 0, 1, 2, or 3. A color number of 0 indicates black and a color number of 1 indicates white. A color number of 2 will be treated as 0, and a color number of 3 will be treated as 1.

If you do not specify a color number, the default is the graphics foreground (or 1 if the value of *gforeground* has not been specified).

When you display text the character foreground color will be 1 (white) and the character background 0 (black), unless you specify 'inverse video' by the **COLOR** statement (with a 0 value of *tforeground*).

You can also specify an XOR operation between the pixels on the screen and the pixels in memory, by specifying *tforeground* with a value greater than 1.

Any parameter in the **COLOR** statement may be omitted. Omitted parameters assume the old values. Upon initialization default values are:

- *gforeground* = 1 (white)
- *gbackground* = 0 (black)
- *tforeground* = 1 (normal video, no XOR)

That is, if no **COLOR** statement exists in your program, the system assumes:

```
COLOR 1,0,1
```

Example

```
SCREEN 3  
COLOR 0,1,0
```

This selects a black graphics foreground on a white background, inverse video and no XOR.

COM(*n*) Statement

COM(*n*) ON enables, COM(*n*) OFF disables, and COM(*n*) STOP suspends event trapping of communications activity on the specified channel.

COM(*n*) {ON | OFF | STOP }

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is an integer expression that specifies the number of the communications channel. It may be 1 or 2.

To Enable or Disable COM(n) Trapping

IF...	THEN...
a COM(n) ON is executed	communications event trapping is enabled. While trapping is enabled, and if a non-zero line number is specified in the ON COM(n) GOSUB statement, GW-BASIC checks between every statement to see if activity has occurred on the communications channel. If it has, the ON COM(n) GOSUB statement is executed.
a COM(n) OFF is executed	communications event trapping is disabled. If an event takes place, it is not remembered.
a COM(n) STOP is executed	communications event trapping is suspended. If an event occurs it is remembered, and ON COM(n) will be executed as soon as trapping is enabled.
an ON COM(n) GOSUB is executed	communications event trapping is suspended.
an error trap takes place	all trapping is automatically disabled (including ERROR trapping).

Example

```
10 COM(1) ON
```

Enables error trapping of communications activity on channel 1.

COMMON Statement

Defines a common area which is not erased by the CHAINED program, and allows you to pass variables from one program to another. COMMON should only be used in a program.

COMMON *variable*[, *variable*]...

Where

SYNTAX ELEMENT	MEANING
<i>variable</i>	Is the name of a numeric or string variable which is required to be passed to the CHAINED program. For array variables place a set of parentheses "()" after the variable name.

Characteristics

The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning.

Variables specified in COMMON statements are allocated in the common area starting from the beginning and in the order in which they appear in the program.

The CHAINED program need not specify, through the use of COMMON statements, the common variables specified by the CHAINing program. The CHAINED program will use these variables with the same names specified in the CHAINing program. Each type definition statement (DEFINT, DEFSNG, DRFDBL, DEFSTR) referring to common variables, must precede the associated COMMON statements and must be repeated in the CHAINED program.

Common variables must always be initialized within the CHAINing program. Common arrays must be explicitly described by DIM statements in the CHAINing program (but not in the CHAINED program, otherwise a "Duplicate definition" error occurs). The DIM statements must be written before the associated COMMON statements.

Example

```
10 REM PG1
20 COMMON A1,B1,C1,D1$
.
.
.
80 CHAIN "A:PG2"
90 END

10 REM PG2
20 PRINT A1,B1,C1,D1$
.
.
.
120 END
```

The above example shows that the CHAINED program need not specify, through the use of COMMON statements, the common variables specified by the CHAINing program.

In our example the values of the variables A1, B1, C1, and D1\$ in the program PG1 are passed to the CHAINED program PG2, which may display them (see Statement 20).

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
10 REM PG1
20 DEFDBL C
30 COMMON A1,B1,C1,D1$
.
.
90 CHAIN "A:PG2"
100 END

10 REM PG2
20 DEFDBL C
.
.
130 END
```

Each type definition statement (DEFINT, DEFSNG, DEFDBL, DEFSTR) referring to common variables, must precede the associated COMMON statement and must be repeated in the CHAINED program. (Note the statements DEFDBL, both with PG1 and PG2.)

Example

```
10 REM PROGRAM1
20 COMMON A$,B$,C$
30 COMMON A$,A1
.
.
100 END
```

It is not good programming practice to repeat the same variable name (in this case A\$) either in different COMMON statements of the same program, or in the same COMMON statement. In any case multiple definitions are equivalent to a single definition.

Example

```
10 REM PG1
20 DIM A1(15,20)
30 COMMON A1(),B1,C1
```

```
.
.
.
```

```
100 CHAIN "A:PG2"
110 END
```

```
10 REM PG2
```

```
.
.
.
```

```
50 PRINT A1(1,1)
```

```
.
.
.
```

```
90 END
```

This example shows the definition and transfer of an array variable, using the COMMON statement, to the chained program.

Example

```
10 REM mod1
20 A = 1:B = 2
30 COMMON A,B
40 GOTO 60
50 COMMON C
60 CHAIN "mod3"
```

```
10 REM mod2
20 A = 1:B = 2
30 COMMON A
40 GOTO 60
50 COMMON B
60 CHAIN "mod3"
```

```
10 REM mod3
20 PRINT A;B
```

COMMANDS, STATEMENTS AND FUNCTIONS

The **COMMON** statement is a declarative statement, thus it allocates a common area even if control of execution does not pass through it.

For example, when executing program "mod1" an "Illegal function call in 50" is issued, as variable C has not been initialized. When executing program "mod2" instead, program "mod3" is CHAINED: it displays both A and B variables, even if statement 50 of "mod2" is jumped over.

CONT Command

Resumes program execution after a **CTRL BREAK** has been typed or a **STOP** or **END** statement has been executed. **CONT** should only be used in immediate mode.

CONT

Characteristics

Execution resumes at the point where the break occurred. If the break occurred after a prompt from an **INPUT** statement, execution continues with the reprinting of the prompt ("?" or prompt string).

CONT is usually used in conjunction with **STOP** for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with **CONT** or a direct mode **GOTO**, which resumes execution at a specified line number.

CONT may not be used to continue execution after an error has occurred. **CONT** is also invalid if the program has been modified during the break.

Example

```
10 INPUT A,B
20 TEMP = A*B
30 STOP
40 FINAL = TEMP + 300: PRINT FINAL
RUN
? 32, 2.4
Break in 30
Ok
PRINT TEMP
76.8
Ok
CONT
376.8
Ok
```

COS Function

Returns the cosine of the argument.

COS(*numexp*)

Characteristics

The argument *numexp* represents the angle in radians.

The calculation of the COS function is performed in single precision, unless /D is supplied in the GWBASIC command line.

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
10 X = 2 * COS(.4)
20 PRINT X
RUN
1.842122
Ok
```

CSNG Function

Converts any numeric argument to a single precision number.

CSNG(*numexp*)

Remarks

See the CINT and CDBL functions for converting numbers to the integer and double precision data types, respectively.

Example

```
10 A# = 975.3421
20 PRINT A#; CSNG(A#)
RUN
975.3421020507813 975.3421
Ok
```




CSRLIN Function

Returns the current line (row) position of the cursor.


CSRLIN

Characteristics

CSRLIN returns an integer value in the range 1 to 25. To return the current column position use the POS function.

Example

```
10 Y = CSRLIN 'Record current line.
20 X = POS(0) 'Record current column.
30 LOCATE 24,1 :PRINT "HELLO" 'Print HELLO on last line.
40 LOCATE Y,X 'Restore position to old line, column.
```



CVI, CVS, CVD Functions

Converts string values to numeric values.

Syntax 1

CVI(*2-byte-string*)

COMMANDS, STATEMENTS AND FUNCTIONS

Syntax 2

CVS(*4-byte-string*)

Syntax 3

CVD(*8-byte-string*)

Characteristics

These functions are useful for converting strings into numbers when dealing with random files.

CVI converts a *2-byte-string* to an integer.

CVS converts a *4-byte-string* to a single precision number.

CVD converts an *8-byte-string* to a double precision number.

See also "MKI\$, MKS\$, MKD\$" functions, later in this chapter.

Example

```
.  
. .  
. .  
70 FIELD #1,4 AS N$, 12 AS B$, ...  
80 GET # 1  
90 Y = CVS(N$)
```

```
.  
. .  
. .
```

DATA Statement

Creates an internal file, i.e., a sequence of data belonging to the program. Each data item will then be assigned to a program variable by a READ statement. A DATA statement should only be used in a program.

DATA *constant* [, *constant*]...

Where

SYNTAX ELEMENT	MEANING
<i>constant</i>	Is a numeric or string constant. Any numeric format (i.e., integer, hexadecimal, octal, single or double precision) is acceptable for numeric constants. String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons, or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

DATA statements are non-executable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas). Any number of DATA statements may be used in a program.

A DATA statement in a program need not correspond to a specific READ statement. This is because before program execution, a data file is created. It contains all the values of all the DATA statements in the program in line number sequence. When the program is executed, READ takes its values from this file.

The data-type of an entry in the data sequence must correspond to the type of the variable to which it is to be assigned; i.e., numeric variables require numeric constants as data (conversion from one numeric type to another is allowed, for example you may have a single precision floating point constant associated with an integer variable) and string variables require quoted or unquoted strings as data.

A quoted string is required if the string contains commas (e.g. "BIRMINGHAM,") or initial or final blanks (e.g. " BIRMINGHAM").

DATA statements may be re-read from the beginning by use of the RESTORE statement.

Example

```
Ok
10 PRINT "CITY", "STATE", "ZIP"
20 READ C$,S$,Z
30 DATA "BIRMINGHAM,",ALABAMA,12345
40 PRINT C$,S$,Z
RUN
CITY          STATE          ZIP
BIRMINGHAM,  ALABAMA        12345
Ok
```




DATE\$ Function and Statement

Retrieves the date (as a function), or sets the date (as a statement).

Syntax 1: As a function

stringvar = **DATE\$**

Syntax 2: As a statement

DATE\$ = *stringexp*

Characteristics

As a function, the current date is fetched and assigned to the string variable *stringvar* . The DATE\$ function may also be used in any string expression in a LET or PRINT statement.

As a statement, the current date is set. In this case DATE\$ is the target of a string assignment.

The date may also have been set by MS-DOS prior to entering GW-BASIC.

COMMANDS, STATEMENTS AND FUNCTIONS

Remarks

If *stringexp* is not a valid string, a "Type mismatch" error will result. Previous values are retained.

For *stringvar* = DATE\$, DATE\$ returns a 10 character string in the form "*mm-dd-yyyy*" where *mm* is the month (01 to 12), *dd* is the day (01 to 31) and *yyyy* is the year (1980 to 2099).

For DATE\$ = *stringexp*, *stringexp* may be one of the following forms:

"*mm-dd-yy*"
or
"*mm/dd/yy*"
or
"*mm-dd-yyyy*"
or
"*mm/dd/yyyy*"

If the month or day is specified by the use of only one digit, GW-BASIC assumes a 0 (zero) in front of it. If the year is specified by the use of one digit (*y*), GW-BASIC assumes the year to be 200*y*; if two digits are specified (*yy*), the year will be assumed to be 19*yy*.

If any of the values are out of range or missing, an "Illegal function call" error is issued. Any previous date is retained.

Example

```
DATE$ = "01-01-83"  
Ok  
PRINT DATE$  
01-01-1983  
Ok
```

DEF FN Statement

Defines and names a user-written function. A DEF FN statement may only be used in a program.

DEF FN*name*[(*argument*[, *argument*]...)] = *expression*

Where

SYNTAX ELEMENT	MEANING
<i>name</i>	Is a legal variable name. No blanks may be inserted between FN and <i>name</i> and the first character of <i>name</i> must be a letter.
<i>argument</i>	Is a "dummy" variable that is to be replaced by the corresponding argument value when the function is called.
<i>expression</i>	<p>Is an expression that performs the operation of the function.</p> <p>The type of expression must agree with the type (numeric or string) of the function, specified by <i>name</i>.</p>

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

In the DEF FN statement, variable names serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the argument list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the program variable is used.

The variables in the argument list represent, on a one-to-one basis, the argument variables or values that are to be given in the function call.

User-defined functions may be numeric or string. The type of the function is specified by *name*. The type of the expression must match the type of the function, otherwise a "Type mismatch" error occurs. If the function is numeric the value of the expression is forced to that type before the function value is returned.

If a DEF FN statement has not been executed before the function it defines is called, an "Undefined user function" error occurs.

Example

```
.  
.   
.   
400 R=1:S=2  
410 DEF FNAB(X,Y)=X ^ 3/Y ^ 2  
420 T=FNAB(R,S)  
.   
.   
. 
```

Line 410 defines the function FNAB. The function is called in line 420. When executed, the variable T will contain the value $(R \wedge 3)$ divided by $(S \wedge 2)$, that is .25.

DEF SEG Statement

Assigns the starting address of the current "segment" of memory. DEF SEG is usually used in a program.

DEF SEG [= address]

Where

SYNTAX ELEMENT	MEANING
<i>address</i>	Is a numeric expression returning an unsigned integer in the range 0 to 65535. The address specified identifies the segment start address used by BLOAD, BSAVE, PEEK, POKE, DEF USR, and CALL.

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

IF...	THEN...
<i>address</i> is omitted	the current segment is set to GW-BASIC's Data Segment. This is the initial default value.
<i>address</i> is specified	the binary value is shifted left 4 bits (i.e., if <i>address</i> is in hexadecimal a zero is appended) to form the current segment start address.
you enter a value outside the 0-65535 range	<p>an "Illegal function call" error results. The previous value will be retained.</p> <p>This is the only check that GW-BASIC makes. Since whatever address is given, within the range, is considered valid, it is the responsibility of the user to ensure that the <i>address</i> is really valid, i.e. so that subroutines are not called from incorrect addresses and important areas of memory are not overwritten.</p>
you do not separate DEF and SEG by at least one blank	<p>GW-BASIC would interpret DEFSEG as the name of a variable. For instance:</p> <p>100 DEFSEG = 150</p> <p>would assign the value 150 to the variable DEFSEG.</p>

Examples


```
10 DEF SEG = &HB800
```

Set segment to Screen buffer

```
100 DEF SEG
```

Restore segment to GW-BASIC's DS.

Note that in statement 10 the screen buffer is at absolute address B8000 hex, as the last hexadecimal digit is omitted in the DEF SEG statement.



DEF USR Statement

Enables access to a machine language subroutine by specifying the starting address. The subroutine may be subsequently called by the associated USR function. DEF USR is usually used in a program.

DEF USR [*n*] = *offset*

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	May be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <i>n</i> is omitted, DEF USR0 is assumed.
<i>offset</i>	<p>Is an integer expression from 0 to 65535. It specifies the starting address of the subroutine as an offset into the current segment.</p> <p>The current segment is as defined by the most recently executed DEF SEG statement or, if no DEF SEG statement has been executed, is the GW-BASIC Data Segment.</p>

Characteristics

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary. To obtain the starting address of a subroutine, GW-BASIC adds the value of *offset* to the start address of the current segment.

Example

```
100 DEF SEG = 0
...
200 DEF USR0 = 24000
210 X = USR0(Y ^ 2/2.89)
...
```


DEFINT/SNG/DBL/STR Statements

Declare the variable type in accordance with the letter(s) specified . These statements are usually used in a program.

DEF*type* *letter*[- *letter*][, *letter*[- *letter*]]...

Where

SYNTAX ELEMENT	MEANING
<i>type</i>	Is INT, SNG, DBL, or STR. No space should be entered between DEF and INT, SNG, DBL, or STR.
<i>letter</i>	Represents a letter from the alphabet (A-Z)

Characteristics

Any variable names beginning with the letter(s) specified in the range of letters will be considered. The type of variable specified by the type declaration character (% , ! , # , \$) always takes precedence over a DEF*type* statement.

If no type declaration statements are encountered, GW-BASIC assumes all variables without declaration characters are single precision variables. DEF*type* statements must precede the use of the defined variables.

COMMANDS, STATEMENTS AND FUNCTIONS

Examples

10 DEFDBL L-P

All variables beginning with the letters L, M, N, O, and P will be double precision variables.

10 DEFSTR A

All variables beginning with the letter A will be string variables.

10 DEFINT I-N, W-Z

All variable beginning with the letters I, J, K, L, M, N, W, X, Y, Z will be integer variables.

DELETE Command

Erases program lines. (DELETE is usually used in immediate mode.)

DELETE [*linenum1*][- [*linenum2*]]

Where

SYNTAX ELEMENT	MEANING
<i>linenum1</i>	Is the first line to be erased
<i>linenum2</i>	Is the last line to be erased

Characteristics

GW-BASIC always returns to command level after a DELETE is executed. If either *linenum1* or *linenum2* does not exist, an "Illegal function call" error occurs. A period (.) can be used instead of the line number to indicate the current line.

Examples

DELETE 80 Deletes line 80.

DELETE 80-120 Deletes lines 80 through 120, inclusive.

DELETE -80 Deletes all lines up to and including line 80.

DELETE 80- Deletes all lines from line 80 through the end of the program.



DIM Statement

Specifies the array name, the number of dimensions and the subscript upper bound per dimension. The DIM statement may specify one or more arrays. DIM is usually used in a program.

DIM *array* (*list-of-subscripts*)[, *array* (*list-of-subscripts*)]...

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
<i>array</i>	Is a valid array name. Any legal variable name may be used.
<i>list-of-subscripts</i>	Comprises one or more numeric expressions which specify the array dimensions. Each subscript must be separated from the next by a comma. The number of subscripts specifies the number of dimensions, and the value of each specifies the subscript upper bound.

Characteristics

If an array name is used without a corresponding DIM statement, the maximum value of the array's subscript(s) defaults to 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement.

If no DIM is specified, the first reference to an array element in the program will create the array with the specified number of dimensions. For example, if a program statement refers to:

AR1(3,5,10)

Then AR1 is created with 3 dimensions and a default upper bound of 10 for each dimension.

The DIM statement sets all numeric array elements to an initial value of zero and elements of string arrays to null strings.

Theoretically, the maximum number of dimensions allowed in a DIM statement is 255 and the maximum number of elements per dimension is 32767. In reality, however, these numbers are limited by line length and memory size.

If you try to redimension an array without first erasing it, a "Duplicate Definition" error occurs. You must first use the ERASE statement to erase an array before redimensioning it.

Number of Elements per Dimension

IF...	AND IF...	THEN...
no DIM is used	OPTION BASE 0 is set	11 elements (subscripts 0-10 are allowed in each dimension).
	OPTION BASE 1 is set	10 elements (subscripts 1-10 are allowed in each dimension).
DIM is used	OPTION BASE 0 is set	the number of elements in each dimension is calculated by adding 1 to each upper bound subscript.
	OPTION BASE 1 is set	the number of elements in each dimension coincides with each upper bound subscript.

COMMANDS, STATEMENTS AND FUNCTIONS

To Define an Array

YOU MUST...	AND EITHER...	OR...
establish the subscript lower bound	use an OPTION BASE 1 statement	adopt the default OPTION BASE 0.
assign a name to the array	use a DIM statement	refer to an array element within the program.
establish the number of dimensions	use a DIM statement	refer to an array element within the program.
establish the subscript upper bound per dimension	use a DIM statement	refer to an array element within the program. Note: In this case a subscript upper bound of 10 for each dimension is assumed.

Remarks

A DIM statement does not set subscript upper bounds if the statement is jumped over (see last example).

Example

```
10 DIM A(5),B$(20,30,15)
```

Example

```
10 INPUT I
20 DIM ARRAY1(I)
30 FOR K=0 TO I
40 READ ARRAY1(K)
50 NEXT
```

.
.
.

Example

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
```

.
.
.

Example

LIST

```
10 I=1
20 GOTO 40
30 DIM A(50)
40 A(10)=3
50 A(11)=45
```

Ok

RUN

Subscript out of range in 50

Ok

The system displays:

Subscript out of range in 50

when statement 50 is executed, as statement 30 is jumped over and an upper bound of 10 is assumed by default.

DRAW Statement

Draws an object as specified by the contents of a string expression.
(Graphics Mode only)

DRAW *stringexp*

Where

SYNTAX ELEMENT	MEANING
<i>stringexp</i>	Is a string expression defining the sequence of Graphics Macro Language (GML) commands that will draw the object.

Characteristics

The DRAW statement makes a group of graphics statements available which facilitate the drawing of complex figures. A GML command is represented by a single character (i.e. U, D, L, R, E, F, G, H, M, B, N, A, C, S, X, P) or a pair of characters (TA) within the string *stringexp*, followed by one or two integer values that represent the coordinates of a point (in pixels).

A WINDOW instruction does not have any effect on the DRAW statement, since DRAW draws directly in the viewport in screen coordinates.

Movement Commands

Each of the following movement commands begin movement from the current graphics position. This is usually the coordinate of the last graphics point plotted with another GML command, LINE, or PSET. The current position defaults to the center of the screen when a program is RUN.

The GML movement commands specify movements by an integer (n), or by a pair of integers (x, y) in the M command. The actual distance travelled (in pixels) is given by this integer multiplied by the "scale factor", defined via the S command (see below).

COMMAND	ACTION
U [n]	Move up. The number of points moved is $n * \text{scale factor}$ (set by the S command below). If n is omitted 1 is supplied.
D [n]	Move down. The number of points moved is $n * \text{scale factor}$ (set by the S command below). If n is omitted 1 is supplied.
L [n]	Move left. The number of points moved is $n * \text{scale factor}$ (set by the S command below). If n is omitted 1 is supplied.
R [n]	Move right. The number of points moved is $n * \text{scale factor}$ (set by the S command below). If n is omitted 1 is supplied.
E [n]	Move diagonally up and right. The number of points moved is $n * \text{scale factor}$ (set by the S command below). If n is omitted 1 is supplied.

COMMANDS, STATEMENTS AND FUNCTIONS

COMMAND	ACTION
F [<i>n</i>]	Move diagonally down and right. The number of points moved is $n * \text{scale factor}$ (set by the S command below). If n is omitted 1 is supplied.
G [<i>n</i>]	Move diagonally down and left. The number of points moved is $n * \text{scale factor}$ (set by the S command below). If n is omitted 1 is supplied.
H [<i>n</i>]	Move diagonally up and left. The number of points moved is $n * \text{scale factor}$ (set by the S command below). If n is omitted 1 is supplied.
M <i>x,y</i>	Move absolute or relative. If x is preceded by a plus (+) or minus (-), x and y are added to the current graphics position, (move relative). Otherwise, they are absolute coordinates with respect to the origin of the axes (move absolute).
B	Move without plotting any points. B may precede any of the above-mentioned movement commands.
N	Move but return to original position when finished. N may precede any of the above-mentioned movement commands.

Further GML Commands

COMMAND	ACTION
A <i>n</i>	Set angle <i>n</i> . <i>n</i> may range from 0 to 3, where 0 is 0 degrees, 1 is 90, 2 is 180, and 3 is 270.
TA <i>n</i>	Rotate angle <i>n</i> . <i>n</i> is expressed in degrees in the range -360 to 360. If <i>n</i> is positive, rotation is counter-clockwise, if <i>n</i> is negative, rotation is clockwise. If <i>n</i> is outside the specified range, an "Illegal function call" error occurs.
C <i>n</i>	Set color number <i>n</i> (from 0 to 3 in Medium Resolution, and 0 to 1 in High or Super Resolution).
Sk	Set scale factor. <i>k</i> may range from 1 to 255. The scale factor is defined as $k/4$. The scale factor, multiplied by the distances given with U,D,L,R,E,F,G,H or relative M commands gives the actual distance travelled (in pixels). If the S command is omitted, a scale factor of 1 (i.e. $k=4$) is assumed.
X <i>stringexp</i>	Execute substring. This powerful command allows you to execute a second substring from a string. You can have one string execute another, which executes a third and so on. Spaces are ignored in <i>stringexp</i> ; but can be inserted for legibility.

COMMANDS, STATEMENTS AND FUNCTIONS

COMMAND	ACTION
P <i>n</i> , <i>m</i>	<i>n</i> is the color number chosen to paint the interior of the closed figure and <i>m</i> is the border color number. You must specify both parameters or an error will occur. Both parameters can range from 0 to 3 in Medium Resolution and from 0 to 1 in High or Super Resolution mode.

Remarks

In all GML commands, numeric arguments can be constants like "327" or `=numvar`; where *numvar* is the name of a numeric variable. The semicolon is necessary if you enter a variable this way or if you use the X command, otherwise you can omit the semicolon between commands.

On the screen, the number of horizontal pixels differs from the number of vertical pixels. The spacing between the pixels also differs. Therefore, in order to draw lines of equal length along the x- and y-axes, you must multiply the vertical points by the aspect ratio (a fraction that is used to take into account the difference in the horizontal and vertical lengths of the screen). For a standard monitor the aspect ratio is 4/3. You must bear in mind these considerations in order to maintain the horizontal and vertical proportions of lines in a figure.

The following program draws a square in Medium Resolution:

```
10 SCREEN 1
20 A = 40
30 B = 48
40 DRAW "U=A; R=B; D=A; L=B;"
```


In Medium Resolution there are 320 horizontal pixels and 200 vertical pixels, therefore 16 horizontal points are equal in length to $10 \times \frac{4}{3}$ vertical points, and, consequently, a line of 48 horizontal points equals a line of 40 vertical points.

Example 1

```
10 U$ = "U30;" : D$ = "D30;" : L$ = "L40;" : R$ = "R40;"
20 BOX$ = U$ + R$ + D$ + L$
30 DRAW "XBOX;"
40 REM DRAW "XU$;XR$;XD$;XL$;" would have drawn the same
   box
```

Example 2

```
10 DRAW "U50R50D50L50" 'Draw a box
20 DRAW "BE10" 'Move up and right into box
30 DRAW "P1,3" 'Paint interior
```

Example 3

```
10 FOR D=0 TO 360 'Draw some spokes
20 DRAW "TA=D;NU50"
30 NEXT D
```

EDIT Command

Lets you change a program line. EDIT is only used in immediate mode.

EDIT {*linenum* | . }

Where

SYNTAX ELEMENT	MEANING
<i>linenum</i>	Is a program line number. If no such line exists, an "Undefined line number" error message is displayed.
	Alternatively a period can be used instead of a line number to specify the current line.

Characteristics

When you enter an EDIT command, GW-BASIC displays the specified line and positions the cursor under the first digit of the line number. The line may then be modified by using the special editor keys.

The EDIT command can be used to redisplay and edit a line which has just been entered. (The line number symbol "." can be specified in this case).

You can also use the LIST command to display a group of program lines for editing.

For examples see Chapter 2.



END Statement

Terminates program execution, closes all open data files, and returns to command level. END is only used in a program.

END

Characteristics

END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a "Break in line *nnnnn*" message to be printed. An END statement at the end of a program is optional. GW-BASIC always returns to command level after an END is executed.

Example

```
520 IF K > 1000 THEN END ELSE GOTO 20
```

ENVIRON Statement

Allows modification of parameters in GW-BASIC's Environment String Table.

ENVIRON *parm*

Where

SYNTAX ELEMENT	MEANING
<i>parm</i>	<p>Is a valid string expression containing the new Environment String parameter.</p> <p>The string expression must only include uppercase letters.</p>

Characteristics

The ENVIRON statement may be used, for example, to change the "PATH" parameter for a child process. Parameters may also be passed to a child process by inventing a new environment parameter.

Remarks

1. *parm* must be of the form *parm-id* = *text* where:
 - a. *parm-id* is the name of the parameter such as "PATH".
 - b. *parm-id* must be separated from *text* by ' = ' or ' ' (blank) such as "PATH = ". ENVIRON takes everything to the left of the first blank or " = " as the *parm-id*, and everything to the right as *text*.
 - c. *text* is the new parameter text. If *text* is a null string, or consists only of ";" (a single semi-colon, such as "PATH = ;") then the parameter (including *parm-id* =) is removed from the Environment String Table and the Table is compressed.
2. If *parm-id* does not exist in the Environment String Table, then *parm-id* is added at the end of the Environment String Table.
3. If *parm-id* does exist, it is deleted, the Environment String Table is compressed and the new *parm-id* is added at the end.

Examples

The following MS-DOS command will create a default "PATH" to the Root Directory on disk A:

```
PATH = A:
```

The PATH may be changed in GW-BASIC to a new value by:

```
ENVIRON "PATH = A:SALES;A:ACCOUNTS"
```

A new parameter may be added to the Environment String Table:

```
ENVIRON "SESAME = PLAN"
```

The Environment String Table now contains:

```
PATH = A:SALES;A:ACCOUNTS  
SESAME = PLAN
```

If you then entered:

```
ENVIRON "SESAME = ;"
```

then you would have deleted SESAME, and you would have a table containing:

```
PATH = A:SALES;A:ACCOUNTS
```

Possible errors

"Type mismatch": if *parm* is not a string.

"Out of Memory": if the Environment Table is full and no more can be allocated.

ENVIRON\$ Function

Allows you to retrieve the specified Environment String from GW-BASIC's Environment String Table.

```
ENVIRON$ {(parm )} {( nth-parm ) }
```

Where

SYNTAX ELEMENT	MEANING
<i>parm</i>	Is a string expression containing the parameter to be retrieved. The string expression must only include uppercase letters.
<i>nth-parm</i>	Is an integer expression returning a value in the range 1 to 255.

Remarks

1. If a string argument is used, ENVIRON\$ returns a string containing the text following *parm=* from the Environment String Table.
2. If *parm=* is not found, or no text follows *parm=* then a null string is returned.
3. If a numeric argument is used, ENVIRON\$ returns a string containing the *nth-parm* from the Environment String Table including the *parm*.
4. If there is no *nth-parm* , then a null string is returned.

Possible errors

- "Illegal function call": if *nth-parm* is out of range.
- "Type mismatch": if *parm* is not a string.
- "String too long": if the string is longer than 255 characters.

COMMANDS, STATEMENTS AND FUNCTIONS

EOF Function

Indicates that the end of a file has been reached. EOF is usually used in a program.

EOF (*filenum*)

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is the file number specified in the OPEN statement

Characteristics

Returns -1 (true) if the end of a sequential file has been reached. Use EOF to test for end-of-file while inputting, to avoid "Input past end" errors.

When EOF is used with random access files, it returns "true" if the last executed GET statement was unable to read an entire record because of an attempt to read beyond the end.

When EOF is used with a communications device, the definition of the end-of-file condition is dependent on the mode (ASCII or binary) in which the device was opened. In binary mode, EOF is true when the input queue is empty ($LOC(n) = 0$). It becomes false when the input queue is not empty. In ASCII mode, EOF is false until a **CTRL Z** is received, and from then on it will remain true until the device is closed.

Example

```
5 DIM M(500)
10 OPEN "I",1,"DATA"
20 K=0
30 IF EOF(1) THEN 100
40 INPUT #1, M(K)
50 K=K+1:GOTO 30
.
.
.
```

This example reads data from the sequential file named "DATA". Values are read into array M until the end of file is reached.

ERASE Statement

Releases space and variable names previously reserved for arrays. The data is lost and the array(s) no longer exist.

ERASE *array* [, *array*]...

Where

SYNTAX ELEMENT	MEANING
<i>array</i>	Is the name of an array to be erased.

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

Arrays may be re-dimensioned after they are ERASEd, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASEing it, a "Duplicate Definition" error occurs.

It is not good programming practice to reuse an identifier. This may generate errors or reduce the program readability. You may, however, find it useful to redeclare an erased array; for example, when an array name is known by a subroutine and you want to pass arrays with different number of dimensions of subscript upper bounds to this subroutine.

Example

```
10 DIM A(15,15),B(10,20)
```

```
100 ERASE A,B
```

```
110 DIM A (100),B(2,2,2)
```

Upon execution of statement 100, arrays A and B are deleted and the corresponding memory space is made free. You may define other arrays (see statement 110) with the same names but different numbers of dimensions and upper bounds.

ERDEV and ERDEV\$ Functions

ERDEV is an integer function which contains the error code returned by the last device to declare an error.

ERDEV\$ is a string function which contains the name of the device driver which generated the error.

{ ERDEV | ERDEV\$ }

Characteristics

ERDEV is set by the Interrupt X'24' handler, when an error within MS-DOS is detected. ERDEV will contain the INT 24 error code in the lower 8 bits, and the upper 8 bits will contain the "Word attribute bits" (b15-b13) from the Device header block.

If the error was on a character device, ERDEV\$ will contain the 8-byte character device name. If the error was not on a character device, ERDEV\$ will contain the two character block device name (A:, B:, C: etc).

Remarks

For the sake of compatibility between different releases, it is advisable to perform error checking by using ERDEV rather than ERDEV\$.

Example

If a user installed device driver "MYLPT2" caused a "Out of Paper" error via INT 24, and the driver's error number for that problem was 9, ERDEV will contain the error number 9 in the lower 8 bits and the device header word attributes in the upper 8 bits, and ERDEV\$ will contain "MYLPT2".

ERR and ERL Functions

The ERR function returns the error code and the ERL function returns the number of the line which contains the error.

{ ERR | ERL }

Characteristics

When an error handling routine is entered, the function ERR contains the error code and the function ERL contains the line number of the line in which the error was detected.

The ERR and ERL functions are usually used in IF...THEN statements to direct program flow in the error handling routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535.

If the line number is not on the right side of the relational operator, it cannot be renumbered with RENUM. Because ERL and ERR are reserved functions, neither may appear to the left of the equal sign in a LET (assignment) statement.

GW-BASIC error codes are listed in Appendix A.

To test whether an error occurred in a direct statement, use:

IF 65535 = ERL THEN

Otherwise, use:

IF ERR = *error-code* THEN ...

IF ERL = *linenum* THEN ...

Example

LIST

```
10 REM RECTANGLE2
20 ON ERROR GOTO 70
30 INPUT "Length and Width";L,W
40 IF (L<0) OR (W<0) THEN ERROR 200
50 PRINT "Area = ";L*W;" L = ";L;" W = ";W
60 GOTO 30
70 IF (ERR=200) AND (ERL=40)
   THEN PRINT "L or W<0":RESUME 30
80 ON ERROR GOTO 0
90 END
Ok
```

RUN

```
Length and Width? -2,5
L or W<0
Length and Width? 2,5
Area = 10 L = 2 W = 5
Length and Width? ^C
Break in 30
Ok
```

If you enter a negative value for L or W, the error handling routine is activated and the system displays:

L or W <0

Execution is resumed at statement 30 (see RESUME statement below). Note the use of ERR and ERL functions in the error handling routine.

ERROR Statement

Simulates the occurrence of a GW-BASIC error, or generates a user defined error.

ERROR *n*

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is an integer expression representing an error code. It must be greater than 0 and less than or equal to 255. If it is not an integer, it is rounded to the nearest integer.

Characteristics

IF...	THEN...
<p>the value of n equals an error code already in use by GW-BASIC</p>	<p>the ERROR is simulated, and the corresponding error message will be displayed.</p> <p>For example:</p> <pre> LIST 10 S = 10 20 T = 5 30 ERROR S + T 40 END Ok RUN String too long in line 30 Ok </pre> <p>Or, in immediate mode:</p> <pre> ERROR 15 String too long Ok </pre>
<p>the value of n is greater than any error codes used by GW-BASIC</p>	<p>the ERROR statement will generate a user-defined error. This user-defined error code may then be handled in the error trapping routine (see the ON ERROR statement in this chapter).</p> <p>Note: To define your own error, use a value that is greater than any used by GW-BASIC error codes. (It is preferable to use the highest available values, so compatibility may be maintained if more error codes are added to GW-BASIC).</p>

COMMANDS, STATEMENTS AND FUNCTIONS

IF...	THEN...
an ERROR statement specifies a code for which no error message has been defined	GW-BASIC responds with the message: Unprintable error

EXP Function

Returns e (base of natural logarithms) to the power of the argument.

EXP(*numexp*)

Remarks

numexp must be ≤ 87.3365 . If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

EXP is calculated in single precision, unless /D is supplied in the GWBASIC command line.

Example

```
10 X = 5
20 PRINT EXP(X-1)
RUN
54.59815
Ok
```


FIELD Statement

Allocates space for variables in a random file buffer. FIELD is usually used in a program.

FIELD [#] *filenum* , *width* **AS** *stringvar* [, *width* **AS** *stringvar*]...

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is the number under which the file was OPENed
<i>width</i>	Is the number of characters to be allocated to <i>stringvar</i>
<i>stringvar</i>	Is a string variable name that will be used for random file access

Characteristics

Before a GET statement or PUT statement can be executed, a FIELD statement must be executed to format the random file buffer.

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 128 bytes.)

COMMANDS, STATEMENTS AND FUNCTIONS

Any number of FIELD statements may be executed for the same file. All FIELD statements that have been executed will remain in effect at the same time.

Remarks

Do not use a FIELDed variable name in an input statement or to the left of the equal sign in an assignment statement.

Once a variable name is FIELDed, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name on the left side of the equal sign is executed, the variable no longer refers to the random file buffer, but to the variables stored in string space.

Note that a variable name, previously defined in a FIELD statement, may be inserted to the right of the equal sign in an assignment statement.

Example 1

```
10 FIELD# 1,20 AS N$,10 AS ID$,40 AS ADD$
```

Allocates the first 20 positions (bytes) in the random file buffer to the string variable N\$, the next 10 positions to ID\$, and the next 40 positions to ADD\$. FIELD does NOT place any data in the random file buffer.

Example 2

```
10 OPEN "R", #1, "A:PHONELST", 35
15 FIELD #1,2 AS RECNBR$, 33 AS DUMMY$
20 FIELD #1,25 AS NAME$, 10 AS PHONENBR$
25 GET #1
30 TOTAL = CVI(RECNBR$)
35 FOR I = 2 TO TOTAL
40 GET #1, I
45 PRINT NAME$, PHONENBR$
50 NEXT I
```

Illustrates a multiple defined FIELD statement. In statement 15, the 35 byte field is defined for the first record to keep track of the number of records in the file; statement 20 defines the field for individual names and phone numbers.

Example 3

```
10 FOR LOOP%=0 TO 7
20 FIELD #1,(LOOP%*16) AS OFFSET$,16 AS A$(LOOP%)
30 NEXT LOOP%
```

Shows the construction of a FIELD statement using an array of elements of equal size. The result is equivalent to the single declaration:

```
FIELD #1,16 AS A$(0),16 AS A$(1),...,16 AS A$(6),16 AS A$(7)
```

Example 4

```
10 DIM SIZE%(4%): REM ARRAY OF FIELD SIZES
20 FOR LOOP%=0 TO 4%:READ SIZE% (LOOP%): NEXT
  LOOP%
30 DATA 9,10,12,21,41

120 DIM A$(4%): REM ARRAY OF FIELDDED VARIABLES
130 OFFSET%=0
140 FOR LOOP%=0 TO 4%
150 FIELD #1,OFFSET% AS OFFSET$,SIZE%(LOOP%) AS
  A$(LOOP%)
160 OFFSET%=OFFSET%+SIZE%(LOOP%)
170 NEXT LOOP%
```

Creates a field in the same manner as Example 3. However, the element size varies with each element. The equivalent declaration is:

```
FIELD #1,SIZE%(0) AS A$(0),SIZE%(1) AS A$(1),...
SIZE%(4%) AS A$(4%)
```

COMMANDS, STATEMENTS AND FUNCTIONS

Example 5

```
10 FIELD # 1,225 AS TST$
```

Note that you must observe the maximum length restriction for various variables. For example in the FIELD statement above the maximum length of TST\$ is 255.

FILES Command

Displays the names of files residing on the specified directory.

FILES [*filespec* | *pathname*]

Where

SYNTAX ELEMENT	MEANING
<i>filespec</i> (or <i>pathname</i>)	Is a string expression specifying either a filename or a directory name, and optionally a drive. If the drive is omitted, the MS-DOS default drive is assumed. If no directory is specified, the current directory for the specified drive is assumed. If only FILES is specified, all the files on the current directory of the MS-DOS default drive will be listed.

Characteristics

A filename may contain question marks (?) or asterisks (*) used as wild cards. A question mark will match any single character in the filename or extension. An asterisk will match one or more characters starting at the position. The asterisk is a shorthand notation for a series of question marks. The asterisk need not be used in the case where all the files on a drive are requested, e.g., FILES "B:".

Examples

FILES

Shows all files on the current directory

FILES "*.BAS"

Shows all files with an extension of .BAS

FILES "A:*.*)"

Shows all files on drive A

FILES "A:"

Equivalent to the preceding example

FILES "GEO?.BAS"

Shows all files on the current directory of the MS-DOS default drive that have a filename of 4 characters beginning with GEO and an extension of .BAS

Sub-directories are denoted by <DIR> following the directory name.

FILES ".\SALES"

If SALES is a subdirectory of the current directory, this command displays SALES <DIR>. If SALES is a file in the current directory, this command displays SALES.

FILES "\SALES\MARY"

Displays MARY <DIR> if MARY is a subdirectory of SALES or, if MARY is a file, displays its name.

FIX Function

Returns the truncated integer part of the argument.

FIX(*numexp*)

Characteristics

FIX(*numexp*) is equivalent to **SGN**(*numexp*)***INT**(**ABS**(*numexp*)). The major difference between **FIX** and **INT** is that **FIX** does not return the next lower number for a negative argument.

Examples

PRINT FIX(58.75)

58

Ok

PRINT FIX(-58.75)

-58

Ok

FOR...NEXT Statements

Allow a series of statements to be performed in a loop a specified number of times. FOR/NEXT are usually used in a program.

FOR *control-variable* = *initial-value* **TO** *final-value* [**STEP** *increment*]

.
.
.
[*loop statements*]
.
.
.

NEXT [*control-variable*][, *control-variable*] ...

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
<i>control-variable</i>	Is an integer or single-precision variable used as a counter
<i>initial-value</i>	Is a numeric expression specifying the first value assigned to the <i>control-variable</i>
<i>final-value</i>	Is a numeric expression specifying the limit of the <i>control-variable</i>
<i>increment</i>	Is a numeric expression specifying the value to be added (with its algebraic sign) to the <i>control-variable</i> when the NEXT statement is encountered

Characteristics

The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the *control-variable* is adjusted by the amount specified by *increment*. A check is performed to see if the value of the *control-variable* is now greater than the *final-value*. If it is not greater, the program branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop.

If STEP is not specified, the *increment* is assumed to be one. If *increment* is negative, the *final-value* must be less than the *initial-value*. The *control-variable* is decreased each time through the loop. The loop is executed until the *control-variable* is less than the *final-value*.

The *control-variable* must be an integer or single precision numeric variable. If a double precision numeric variable is used, a "Type mismatch" error will result.

The body of the loop is skipped if either:

- the *increment* is positive, and the *initial-value* exceeds the *final-value*
- the *increment* is negative, and the *initial-value* is less than the *final-value*

Nested Loops

FOR...NEXT loops may be nested; that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement (with a list of control variables) may be used for all of them.

Note that a statement of this form:

```
NEXT V1, V2, V3
```

performs the same action as the sequence of statements:

```
NEXT V1  
NEXT V2  
NEXT V3
```

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. When using nested loops, the variable(s) in each NEXT statement must be specified.

If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

COMMANDS, STATEMENTS AND FUNCTIONS

Delay Loops

Between two statements, in a program, you can insert a dummy FOR/NEXT loop (i.e. without statements between FOR and NEXT) in order to delay the execution of the second statement. This can prove useful in various situations, for example in animation.

Example 1

```
10 K = 10
20 FOR I = 1 TO K STEP 2
30 PRINT I;
40 K = K + 10
50 PRINT K
60 NEXT
```

RUN

```
1 20
3 30
5 40
7 50
9 60
```

Ok

Note that although the value of K is changed within the loop, the *final-value* remains as 10.

Example 2

```
10 J = 0
20 FOR I = 1 TO J
30 PRINT I
40 NEXT I
```

RUN

Ok

In this example, the loop does not execute because the *initial-value* of the loop exceeds the *final-value*.

Example 3

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1 2 3 4 5 6 7 8 9 10
Ok
```

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set.

FRE Function

Returns the number of bytes in memory not being used by GW-BASIC.

FRE(*dummy*)

Characteristics

The argument to FRE is a dummy argument. Any value may be supplied. FRE("") forces a garbage collection before returning the number of free bytes.

GW-BASIC will not initiate garbage collection until all free memory has been used up. Therefore, using FRE("") periodically will result in shorter delays for each garbage collection.

Example

```
PRINT FRE(0)
14542
Ok
```

GET (COM files) Statement

Reads a specified number of bytes from the communications buffer. GET(COM files) is usually used in a program.

GET [#] *filenum* , *length*

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is an integer expression returning a file number.
<i>length</i>	Is an integer expression returning the number of bytes to be transferred into the communications buffer. <i>length</i> cannot be greater than the value specified by the LEN clause in the OPEN COM statement.

Example

100 GET # 2,80

GET (Files) Statement

Reads a record from a random disk file into a random buffer. GET (Files) is usually used in program.

GET [#] *filenum* [, *recordnum*]

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is the number under which the file was OPENed
<i>recordnum</i>	Is the number of the record to be read, in the range 1 to 16,777,215. If it is omitted, the next record (after the last GET) is read into the buffer.

Characteristics

The largest possible record number is 16,777,215. This permits large files with short record lengths.

After a GET statement has been executed, you can either refer to FIELDed variables or use INPUT # or LINE # to read characters from the random file buffer. The EOF function may be used after a GET statement to see if that GET was beyond the end of file marker.

COMMANDS, STATEMENTS AND FUNCTIONS

Example

LIST

```
10 OPEN "r",1,"A:RAND",48
20 FIELD 1,20 AS R1$,20 AS R2$,8 AS R3$
30 FOR L=1 TO 4
40 GET 1,L
50 PRINT R1$,R2$,R3$
60 NEXT
70 CLOSE 1
80 END
```

Ok

RUN

super man	USA	11234621
robin hood	England	23462101

.


.

.

Ok

This program retrieves information stored in the specified file. The data read into the buffer may be accessed by the program. This is done here by a PRINT statement (statement 50). See the PUT (files) statement.

GET (Graphics) Statement



Transfers to an array a graphic image contained within a rectangular area of the screen.

GET [STEP] (x1 , y1) - [STEP] (x2 , y2), array

Where

SYNTAX ELEMENT	MEANING
$(x1,y1)-(x2,y2)$	Define a rectangular area on the display screen. $x1,y1$ are the upper left and $x2,y2$ the lower right coordinates. They may be given in absolute or relative form (if the STEP option is used).
<i>array</i>	Is the name assigned to the array that will hold the image, bounded by the specified rectangle.

Characteristics

The GET statement transfers the screen image bounded by the rectangle described by the specified points into the array. The images can be transferred back to any position on the screen using the PUT statement. GET and PUT permit animation and high-speed object motion.

The array must be numeric, but may be any precision. It must be dimensioned large enough to hold the entire image. It is advisable to use arrays of integer type; other types of array can return meaningless values after a GET statement is executed.

Array Dimensions

The storage format in the array is as follows:

- 2 bytes containing the x dimension of the rectangular area (in pixels)
- 2 bytes containing the y dimension of rectangular area (in pixels)
- The array data itself

COMMANDS, STATEMENTS AND FUNCTIONS

Each line of pixels within the rectangular area, starting from the top line, is scanned from the left-hand margin; the color numbers for 8 consecutive pixels are stored in the array at intervals. If there are less than a multiple of 8 pixels in a line being stored, the array element corresponding to the last pixels of the line, will be padded out with zeros. The required array size, in bytes, is:

$$4 + \text{INT}((x * \text{bitsperpixel} + 7) / 8) * y$$

Where

bitsperpixel is 2 in Medium Resolution, and 1 in High and Super Resolution.

The bytes per element of an array type are:

- 2 for integer
- 4 for single precision
- 8 for double precision

Example

For example if you want to GET a 10 by 12 image into an integer array, in Medium Resolution mode, the number of bytes required is $4 + \text{INT}((10 * 2 + 7) / 8) * 12$ or 40 bytes. So, you would need an integer array with at least 20 elements.

It is possible to examine the *x* and *y* dimensions and even the data itself if an integer array is used. The *x* dimension is in element 0 of the array, and the *y* dimension is found in element 1. It must be remembered, however, that integers are stored low byte first, then high byte, but the data is transferred high byte first (leftmost) and then low byte.

Example

```
10 CLS : SCREEN 3 : PSET(20,20)
20 X$ = "R20D20L20U20" : DRAW X$
30 DIM BOX%(64) : GET(20,20)-(40,40),BOX%
35 PRINT "PRESS A KEY" : A$ = INPUT$(1)
40 CLS : PUT(100,100),BOX%
```

GOSUB...RETURN Statements

GOSUB transfers control to a GW-BASIC subroutine by branching to the specified line. RETURN transfers control to the statement following the most recent GOSUB (or ON...GOSUB) executed, or to a specified line. GOSUB/RETURN are only used in a program.

GOSUB *linenum1*

.

.

.

RETURN [*linenum2*]

Where

SYNTAX ELEMENT	MEANING
<i>linenum1</i>	Is the first line number of the subroutine
<i>linenum2</i>	Is any line of your program different from <i>linenum1</i> and from the line number of the GOSUB statement

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

A subroutine may be called any number of times in a program. A subroutine may also be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine causes GW-BASIC to branch back to the statement following the most recent GOSUB or ON...GOSUB statement executed. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine.

The *linenum2* option may be included in the RETURN statement to return to a specific line number from the subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the GOSUB will remain active, and errors such as "FOR without NEXT" may result.

Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

If either *linenum1* or *linenum2* does not exist in the program, an "Undefined line number" error is returned.

Example

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

GOTO Statement

Transfers control to a specified program line.

GOTO *linenum*

Where

SYNTAX ELEMENT	MEANING
<i>linenum</i>	Is the number of a line in the program

Characteristics

If *linenum* is the line number of an executable statement, that statement and those following are executed. If it is the line number of a nonexecutable statement, execution proceeds at the first executable statement encountered after *linenum*. If the specified *linenum* does not exist in the program, an "Undefined line number" error is returned.

COMMANDS, STATEMENTS AND FUNCTIONS

Example

LIST

```
10 READ R
20 PRINT "R = ";R,
30 A=3.14*R ^ 2
40 PRINT "AREA = ";A
50 GOTO 10
60 DATA 5,7,12
Ok
```

RUN

```
R = 5          AREA = 78.5
R = 7          AREA = 153.86
R = 12         AREA = 452.16
?Out of DATA in 10
Ok
```

GW BASIC Command

Initializes GW-BASIC and the operating environment (GW BASIC is an MS-DOS command, not a GW-BASIC command).

GW BASIC [*filespec* | *pathname*] [<*stdin*>] [> *stdout*]
[/F: *number-of-files*] [/S: *lrec*] [/C: *buffer-size*]
[/M: *highest-memory*] [, *max-blocksize*] [/D] [/I]

Where

Options beginning with a slash (/) are called switches. A *switch* is a means used to specify parameters.

SYNTAX ELEMENT	MEANING
<i>filespec</i> (or <i>pathname</i>)	<p>Is a string literal (not included in quotation marks) that specifies a GW-BASIC program file. If the file is present, GW-BASIC proceeds as if a RUN <i>filespec</i> command were given after initialization is complete.</p> <p>A default extension of .BAS is used if none is supplied and the filename is less than 9 characters long. The <i>filespec</i> option allows GW-BASIC programs to be run in batch by putting this form of the command line in an AUTOEXEC.BAT file. GW-BASIC programs which run this way will need to exit via the SYSTEM command in order to allow the next command from the AUTOEXEC.BAT file to be executed.</p>
<i>stdin</i>	<p>Is a literal string (not included in quotation marks) for the standard input file specification. GW-BASIC input is redirected from the file specified by <i>stdin</i>. When present, this parameter must appear before any switches. (See "Re-direction of Standard Input and Output below).</p>
<i>stdout</i>	<p>Is a literal string (not included in quotation marks) for the standard output file specification. GW-BASIC is redirected to the file specified by <i>stdout</i>. When present, this parameter must appear before any switches. (See "Re-direction of Standard Output" below).</p>
<i>/F: number-of-files</i>	<p>Is a switch that sets the maximum number of files (from 1 to 15) that may be open simultaneously during the execution of a GW-BASIC program. It is ignored unless the <i>/I</i> switch is specified on the command line. Refer to the <i>/I</i> switch below.</p>

COMMANDS, STATEMENTS AND FUNCTIONS

SYNTAX ELEMENT	MEANING
	<p>If this switch and the <i>/I</i> switch are present, then the maximum number of files is set to <i>number-of-files</i>. Each file requires 62 bytes for the File Control Block (FCB) plus 128 bytes for the data buffer. The data buffer size may be altered via the <i>/S:</i> switch. If the <i>/F:</i> option is omitted, the <i>number-of-files</i> is set to 3.</p> <p>The number of open files that MS-DOS supports depends upon the value of the <i>FILES = parameter</i> in the CONFIG.SYS file. It is recommended that <i>FILES = 10</i> for GW-BASIC. Remember that the first 3 are taken by <i>stdin</i>, <i>stdout</i>, <i>stderr</i>, <i>stdaux</i>, and <i>stdprn</i>. One additional file handler is needed by GW-BASIC for LOAD, SAVE, CHAIN, NAME and MERGE. This leaves 6 for GW-BASIC File I/O, thus <i>/F:6</i> is the maximum supported by MS-DOS when <i>FILES = 10</i> appears in the CONFIG.SYS file.</p> <p>Attempting to OPEN a file after all the file handlers have been exhausted will result in a "Too many files" error.</p>
<i>/S: lrecl</i>	<p>Is a switch that sets the maximum record length allowed with random files. It is ignored unless the <i>/I</i> switch is specified on the command line. If this switch and the <i>/I</i> switch are present, then the maximum record length is set to <i>lrecl</i>. The record length option (<i>record-length</i>) on the OPEN statement cannot exceed this value. If the <i>/S:</i> option is omitted, the record length defaults to 128 bytes. The maximum value permitted for <i>lrecl</i> is 32767 bytes.</p>

SYNTAX ELEMENT	MEANING
<i>/C: buffer-size</i>	<p>If present, controls RS232 Communications. <i>/C:0</i> disables RS232 support; any subsequent I/O attempts will result in a "Device Unavailable" error. Specifying <i>/C:n</i> allocates <i>n</i> bytes for the receive buffer for each RS232 port present. If the <i>/C:</i> option is omitted, GW-BASIC allocates 256 bytes for the receive buffer of each RS232 port present. 128 bytes are always allocated to the transmit buffer. The maximum value permitted for <i>buffer-size</i> is 32767.</p>
<i>/M: [highest-memory] [,max-blocksize]</i>	<p>When present, <i>highest-memory</i> sets the maximum number of bytes that will be used as GW-BASIC workspace. GW-BASIC will attempt to allocate 64K of memory for the data and stack segment. If machine language subroutines are to be used with GW-BASIC programs, use the <i>/M:</i> switch to set the highest memory location that GW-BASIC can use. When omitted or 0, GW-BASIC attempts to allocate all it can up to a maximum of 65536 bytes.</p> <p>In order to load programs above the GW-BASIC workspace you must use the optional parameter <i>max-blocksize</i> to reserve areas for the workspace and your programs. This is necessary if you intend to use the SHELL command. Failure to do so will result in COMMAND being loaded on top of your routines when a SHELL command is executed.</p> <p><i>max-blocksize</i> must be in Paragraphs (by multiples of 16). When omitted, &H1000 (4096) is assumed. This allocates 65536 bytes ($65536 = 4096 \times 16$) for GW-BASIC's Data and Stack segment. If you require 65536 bytes for GW-BASIC and 512 bytes for machine language subroutines, then use <i>/M:,&H1010</i> (4096 paragraphs for GW-BASIC + 16 paragraphs for your routines).</p>

COMMANDS, STATEMENTS AND FUNCTIONS

SYNTAX ELEMENT	MEANING
	<p>This option can also be used to shrink the GW-BASIC block in order to free more memory for SHELLing other programs. /M:;2048 says: "Allocate and use 32768 bytes maximum for data and stack". /M:32000,2048 allocates 32768 bytes maximum but GW-BASIC will only use the lower 32000. This leaves 768 bytes available for program space.</p>
/D	<p>If present, causes the Double Precision Transcendental math package to remain resident. The functions that will be calculated in double precision if this package is resident are: ATN, COS, EXP, LOG, SIN, SQR, and TAN. If omitted, this package is discarded and the space is freed for program use. The amount of memory required by this package is approximately 3,000 bytes.</p>
/I	<p>GW-BASIC is able to dynamically allocate space required to support file operations. For this reason GW-BASIC does not need to support the /S: and /F: switches. However, some applications are written in such a manner that certain BASIC internal data structures must be static. In order to provide compatibility with these BASIC programs, GW-BASIC will statically allocate space required for file operations based on the /S: and /F: switches when the /I switch is specified.</p>

number-of-files, *lrecl*, *buffer-size*, *highest-memory*, and *max-blocksize* are numbers that may be Decimal, Octal (preceded by &O), or Hexadecimal (preceded by &H).

Examples

A > GWBASIC PAYROLL

Uses 64k of memory and 3 files, loads and executes PAYROLL.BAS.

A > GWBASIC INVENT/F:6/I

Uses 64k of memory and 6 files, loads and executes INVENT.BAS.

A > GWBASIC /C:0/M:32768

Disables RS232 support and uses only the first 32k of memory.

A > GWBASIC /F:4/S:512/I

Uses 4 files and allows a maximum record length of 512 bytes.

A > GWBASIC TTY/C:512

Uses 64k of memory and 3 files, allocates 512 bytes to RS232 receive buffers, load and execute TTY.BAS.

Redirection of Standard Input and Output

Under GW-BASIC you can redirect your Input and Output. Generally, standard input is read from the keyboard, but this can also be the contents of any file specified on the GW-BASIC command line. Standard output, generally to the screen, can be redirected to any device or file specified on the GW-BASIC command line.

Remarks

1. When redirected, all INPUT, LINE INPUT, INPUT\$ and INKEY\$ statements will read from the file specified using *stdin* instead of from the keyboard.

COMMANDS, STATEMENTS AND FUNCTIONS

2. All PRINT and PRINT USING statements will write to the file or peripheral device specified using *stdout* instead of the screen. If a pair of greater than symbols (> >) are entered before *stdout*, data output by PRINT or PRINT USING statements will be "appended" to the specified file.
3. Error messages go to standard output, and appear on the screen.
4. File input from KYBD: still reads from the keyboard.
5. File output to SCRNL: still outputs to the screen.
6. GW-BASIC will continue to trap keys from the keyboard when the ON KEY(n) statement is used.
7. The printer echo key will not cause LPT1: echoing if Standard Output has been re-directed.
8. Typing **CTRL BREAK** will cause GW-BASIC to close any open files, issue the message "Break in line *line-number* " to standard output, exit GW-BASIC, and return to MS-DOS.
9. When input is re-directed, GW-BASIC will continue to read from this source until a **CTRL Z** is detected. This condition may be tested with the EOF function. If the file is not terminated by a **CTRL Z** or if an attempt is made to read past end-of-file by an INPUT # statement, then any open files are closed. The message "Read past end" is then written to standard output, and GW-BASIC returns to MS-DOS.
10. Because of the way in which MS-DOS handles text files it is not recommended to execute a program in GW-BASIC with output rerouted and appended to a file created previously, either with EDLIN or in sequential mode in GW-BASIC: commands such as TYPE will only be able to show you the original contents.

Examples

GWBasic MYPROG > DATA.OUT

Data read by INPUT and LINE INPUT will continue to come from the keyboard. Data output by PRINT goes into the file DATA.OUT.

GWBasic MYPROG < DATA.IN

Data read by INPUT and LINE INPUT will come from DATA.IN. Data output by PRINT goes to the screen.

GWBasic MYPROG < MYINPUT.DAT > MYOUTPUT.DAT

Data read by INPUT and LINE INPUT comes from the file MYINPUT.DAT and data output by PRINT goes into MYOUTPUT.DAT.

GWBasic MYPROG < \SALES\JOHN\TRANS. > > \SALES\SALES.DAT

Data read by INPUT and LINE INPUT will now come from the file \SALES\JOHN\TRANS. Data output by PRINT will be appended to the file \SALES\SALES.DAT.

HEX\$ Function

Returns a string which represents the hexadecimal value of the decimal argument.

HEX\$(*numexp*)

Characteristics

The parameter *numexp* is rounded to an integer before HEX\$ is evaluated. If *numexp* is negative, the two's complement form is used.

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
10 INPUT X
20 A$=HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"
RUN
? 32
32 DECIMAL IS 20 HEXADECIMAL
Ok
```

See the OCT\$ function later in this chapter for details on octal conversion.

IF...GOTO....ELSE and IF...THEN...ELSE Statements

Make a decision regarding program flow based on the result of a specified condition. IF...GOTO...ELSE and IF...THEN...ELSE are usually used in a program.

Syntax 1

IF *condition* **GOTO** *linenum* [**ELSE** {*statements*|*linenum*}]

Syntax 2

IF *condition* **THEN**{*statements* |*linenum* }[**ELSE** {*statements*|*linenum*}]

Where

SYNTAX ELEMENT	MEANING
<i>condition</i>	May be a numeric, relational, or logical expression. GW-BASIC determines whether the condition is true or false by testing the result of the expression for non-zero and zero respectively. A non-zero result is true and a zero result is false. Because of this, you can test whether the value of a variable is non-zero or zero by merely specifying the name of the variable as <i>condition</i> .
<i>statements</i>	Are one or more statements. Each statement must be separated from the preceding one by a colon (:).
<i>linenum</i>	Is a line number of the program in memory

Characteristics

If the result of *condition* is true (not zero), the GOTO or THEN clause is executed. GOTO is always followed by a line number. THEN may be followed by either a line number for branching or one or more statements to be executed. If the result of *condition* is false (zero), the GOTO or THEN clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. A comma is allowed before THEN.

COMMANDS, STATEMENTS AND FUNCTIONS

Nesting of IF Statements

IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example:

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X  
    THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example:

```
IF A = B THEN IF B = C THEN PRINT "A = C"  
    ELSE PRINT "A < > C"
```

will not print "A < > C" when $A < > B$.

If an IF...THEN statement is followed by a line number in direct mode, an "Undefined line" error results, unless a statement with the specified line number had previously been entered in indirect mode.

Remarks

When using IF to test equality for a value that is the result of a floating-point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS(A-1.0) < 1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 1

```
200 IF I THEN GET #1,I
```

This statement GETs record number I, if I is not zero.

Example 2

```
100 IF(I < 20)*(I > 10) THEN DB = 1979-I:GOTO 300
110 PRINT "OUT OF RANGE"
```

·
·
·

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

This statement causes printed output to go either to the terminal or the line printer, depending on the value of the variable IOFLAG. If IOFLAG is zero, output goes to the line printer; otherwise, output goes to the screen.



INKEY\$ Function

Returns either a one- or two-character string read from the keyboard or a null string if no character is pending at the keyboard. INKEY\$ is always used in a program.

INKEY\$

Characteristics

INKEY\$ returns one of the following values:

- A null string if no character is read from the keyboard

COMMANDS, STATEMENTS AND FUNCTIONS

- A one-character string in accordance with a single character read from the keyboard
- A two-character string if a Key (or a Key combination) is entered, that cannot be associated with a standard ASCII code. The first character is hex zero (00); the second indicates the extended code (refer to the "MS-DOS User Guide" for a list of the extended codes)

Although more than one character may be pending in the keyboard buffer, a single character only will be read (unless it is part of an extended code, in which case two characters are read). This value must then be assigned to a variable before it is considered by the GW-BASIC program.

Remarks

No characters are echoed by this statement. All characters are passed to the program, except for the following control characters:

PRTSC	which prints the screen
CTRL NUMLOCK	which sets the system to pause
CTRL BREAK	which stops the program
ALT CTRL DEL	which resets the system

Note that **CR** is passed to the program like any other character.

Example

```
1000 'Timed input Subroutine
1010 RESPONSE$ = ""
1020 FOR I% = 1 TO TIMELIMIT%
1030 A$ = INKEY$:IF LEN(A$) = 0 THEN 1060
1040 IF ASC(A$) = 13 THEN TIMEOUT% = 0:RETURN
1050 RESPONSE$ = RESPONSE$ + A$
1060 NEXT I%
1070 TIMEOUT% = 1:RETURN
```


This subroutine returns two values:

RESPONSE\$ which contains the string entered from keyboard

TIMEOUT% which equals 0 if the user enters a carriage return from the keyboard before a specified number of loops (TIMELIMIT%); otherwise TIMEOUT% equals 1.

Example

The following program will display the INKEY\$ results.

```
10 S$ = INKEY$
20 IF LEN(S$)=0 THEN GO TO 10
30 IF LEN(S$) = 2 THEN GO TO 100
35 ' ... DISPLAYS 1-BYTE CODES ....
40 PRINT HEX$(ASC(S$))
50 GOTO 10
100 ' ... DISPLAYS 2-BYTE CODES ... SECOND BYTE IN HEX
    (AND DECIMAL)
110 S1$ = MID$(S$,1,1): S2$ = MID$(S$,2,1)
120 PRINT HEX$(ASC(S1$)); " "; HEX$(ASC(S2$));
    " (";ASC(S2$);")"
130 GOTO 10
```

INP Function

Returns the byte read from an input port.

INP(*port*)

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
<i>port</i>	Is the input port number in the range 0 through 65535

Remarks

INP is the complimentary function to the OUT statement.

Example

100 A% = INP (54321)

INPUT Statement

Allows input from the keyboard during program execution. INPUT is only used in a program.

INPUT [;] [*prompt* ;]*variable*[, *variable*]...

Where

SYNTAX ELEMENT	MEANING
<i>prompt</i>	Is a string constant enclosed in quotation marks which prompts you for the values you have to enter from the keyboard
<i>variable</i>	Is a numeric or string variable to which is assigned the value entered from the keyboard

Characteristics

When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If *prompt* is included, the string is displayed before the question mark. The required data is then entered at the terminal.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark.

If INPUT is immediately followed by a semicolon, then the CR typed by the user to input data does not echo a CR LF sequence.

The data that is entered is assigned to the variable(s) given in the variable list. The number of data items supplied must be the same as the number of variables in the list. Data items must be separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

COMMANDS, STATEMENTS AND FUNCTIONS

Responding to INPUT with too many or too few items or with the wrong type of value (numeric instead of string, etc.) causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

You may use all the GW-BASIC screen editor features in responding to INPUT and LINE INPUT statements.

Example

```
10 INPUT X
20 PRINT X "SQUARED IS" X ^ 2
30 END
RUN
? 5
5 SQUARED IS 25
Ok
```

Example

```
LIST
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R ^ 2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS? 7.4
THE AREA OF THE CIRCLE IS 171.9464

WHAT IS THE RADIUS?
etc.
```


INPUT # Statement

Reads data items from a sequential disk file and assigns them to program variables. INPUT # is usually used in a program.

INPUT # *filenum* , *variable* [, *variable*]...

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is the number used when the file was OPENed for input
<i>variable</i>	Is a numeric or string variable which will receive a data item from the file. (The type of data in the file must match the type specified by the variable name.) With INPUT # , no question mark is printed, as with INPUT.

Characteristics

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns, and line feeds are ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed, or comma.

COMMANDS, STATEMENTS AND FUNCTIONS

If GW-BASIC is scanning the sequential data file for a string item, leading spaces, carriage returns, and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage return, or line feed (or after 255 characters have been read). If end-of-file is reached when a numeric or string item is being INPUT, the item is terminated.

Example

```
100 INPUT # 1,X$,Y$,Z$
```

This example uses the INPUT # statement to read data from a sequential file into the program.

INPUT\$ Function

Returns a string of characters read from the standard input device, the keyboard, or from a file.

INPUT\$ is usually used in program.

```
INPUT$( length[ , [ # ]filenum] )
```

Where

SYNTAX ELEMENT	MEANING
<i>length</i>	Is an integer expression specifying the number of characters to be read from the keyboard or a file.
<i>filenum</i>	Is the file number specifying the file to be read. If you omit <i>filenum</i> , the keyboard is read by default.

Characteristics

If the keyboard is used for input, no characters will be displayed on the screen. All characters (including control characters) are passed through except **CTRL BREAK** and **CTRL C**, which are used to interrupt the execution of the **INPUT\$** function.

The **INPUT\$** function is preferred over **INPUT #** and **LINE INPUT #** statements, when reading COM files, since all ASCII characters may be significant in communications. **INPUT #** is less desirable because input stops when a comma (,) or **CR** is encountered and **LINE INPUT #** terminates when a **CR** is encountered.

Example 1

```
5 'LIST THE CONTENTS OF A SEQUENTIAL FILE IN
  HEXADECIMAL
10 OPEN "I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,# 1)));
```

COMMANDS, STATEMENTS AND FUNCTIONS

```
40 GOTO 20
50 PRINT
60 END
```

Example 2

```
.
.
.
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$ = INPUT$(1)
120 IF X$ = "P" THEN 500
130 IF X$ = "S" THEN 700 ELSE 100
.
.
.
```

Example 3

This sequence of statements may be used to read a communications file:

```
10 WHILE NOT EOF(1)
20 A$ = INPUT$(LOC(1), #1)
30 ...
40 ... Process data returned in A$ ...
50 ...
60 WEND
```

The above sequence of statements read: "... While there is something in the output queue, return the number of characters in the queue and store them in A\$. If there are more than 255 characters, only 255 will be returned at a time to prevent "String Overflow". Further, if this is the case, EOF(1) is false and input continues until the input queue is empty.."

INSTR Function

Searches for the first occurrence of a given substring in a string, and returns the position at which the match is found.

INSTR([*start* ,]*string* , *substring*)

Where

SYNTAX ELEMENT	MEANING
<i>start</i>	Is an integer expression in the range 1 to 255, which specifies where the search is to begin. If omitted, 1 is assumed.
<i>string</i>	Is a string expression (in particular a string constant or variable) whose value is the string to be searched
<i>substring</i>	Is a string expression (in particular a string constant or variable) whose first occurrence is to be searched for

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

IF...	THEN...
<i>start</i> > LEN (<i>string</i>)	the returned value is 0
<i>start</i> < 1 or <i>start</i> > 255	an error is returned (Illegal function call)
<i>string</i> is empty (null string)	the returned value is 0
<i>substring</i> cannot be found	the returned value is 0
<i>substring</i> is empty and <i>start</i> is specified	the returned value is <i>start</i> .
<i>substring</i> is empty and <i>start</i> is omitted	the returned value is 1

Example

```
10 X$ = "ABCDEB"  
20 Y$ = "B"  
30 PRINT INSTR(X$,Y$);INSTR(4,X$,Y$)  
RUN  
2 6  
Ok
```

Note that the position at which the match is found is always evaluated from the beginning of the original string, even if *start* is specified.

INT Function

Returns the largest integer that is equal to, or less than the argument.

INT(*numexp*)

Characteristics

Refer to the CINT and FIX functions, which also return integer values.

Examples

PRINT INT(99.89)

99

Ok

PRINT INT(-12.11)

-13

Ok

IOCTL Statement

Sends a "Control Data" string to a Character Device Driver anytime after the Driver has been OPENed. IOCTL is usually used in a program.

COMMANDS, STATEMENTS AND FUNCTIONS

IOCTL [#]*filenum* , *string*

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is the file number open to the Device Driver
<i>string</i>	Is a string expression containing the Control Data

Characteristics

IOCTL commands are generally 2 to 3 characters optionally followed by an alphanumeric argument. An IOCTL command string may be up to 255 bytes long.

The IOCTL statement works only if:

1. The device driver is installed.
2. The device driver processes IOCTL strings.
3. GW-BASIC performs an OPEN on a file on that device.

Most standard MS-DOS device drivers don't process IOCTL strings, and it is necessary for the programmer to determine whether the specific driver can handle the command.

If a user has installed his own Driver to replace LPT1, and that Driver is able to set Page Length, then an IOCTL command to set or change the page length might be:

PL_n

where *n* is the new page length.

Example

Opening the new LPT1 driver and setting the Page Length to 66 lines would then be:

```
10 OPEN "LPT1:" FOR OUTPUT AS #1
20 IOCTL #1,"PL66"
```

Example

```
10 OPEN "\\DEV\LPT1" FOR OUTPUT AS #1
20 IOCTL #1,"PL56"
```

Also open LPT1, but with an initial Page Length of 56 lines.

Other user definable IOCTL commands are possible, such as, PT*n* (set Print Tabs every *n* spaces).

Possible Errors

"Bad file number"	- IOCTL to a Driver that is not OPEN.
"Illegal function call"	- If Device does not support IOCTL.
"Device Fault"	- If error in Control Data.

IOCTL\$ Function

Returns a "Control Data" string from a Character Device Driver that is OPEN. IOCTL\$ is usually used in a program.

IOCTL\$([#]*filenum*)

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is the file number open to the device

Characteristics

The IOCTL\$ function is most frequently used to receive acknowledgment that an IOCTL statement succeeded or failed, or to obtain current status information.

IOCTL\$ could be used to ask a communications device to return the current baud rate, information on the last error, logical line width, etc.

The IOCTL\$ function works only if:

1. The device driver is installed.
2. The device driver processes IOCTL strings.

3. GW-BASIC performs an OPEN on a file on that device.

Also see the IOCTL statement.

Example

```
10 OPEN "\DEV\FN1" AS #1
20 IOCTL #1,"RAW" 'Tell device that data is "RAW"
30 IF IOCTL$(1) = "0" THEN CLOSE 1
```

If Character Driver FN1 gives a false return from the Raw data mode IOCTL request, then close the files and stop processing.

Possible Errors

"Bad file number" - IOCTL to a Driver that is not OPEN.
"Illegal function call" - If Device does not support IOCTL.



KEY Statement

Sets a function key to automatically type any sequence of characters. Other options allow you to enable or disable the function key display from the 25th line, or to list the function key values.

KEY{ OFF|ON|LIST|*n* , *stringexp*}

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is the 'key number'. An expression returning an unsigned integer in the range 1 to 10
<i>stringexp</i>	Is a string expression assigned to the key. String constants should be enclosed in quotation marks. The <i>stringexp</i> value may be up to 15 characters long. Longer strings are truncated to 15 characters.

Remarks

The KEY statement enables the designation of a function key as a *softkey*. This means that you can set any function key to generate any sequence of characters.

STATEMENT	MEANING
KEY OFF	Erases the soft key display from the bottom line, making this line available for your GW-BASIC program. In this case you can use LOCATE 25,1 followed by PRINT to display data on the bottom line of the screen. KEY OFF does not disable the function keys.

STATEMENT	MEANING										
KEY ON	Causes the softkey values to be displayed on the bottom line of the screen. If the screen width is 80, all ten softkeys are displayed, but only five softkeys are displayed if the width is 40. In either case, only the first 6 characters of each key value are displayed. If fewer than the total number of function keys are displayed, the user may scroll the function key display (increasing the number of the leftmost key displayed by one each time) by pressing CTRL T . ON is the default state.										
KEY LIST	Displays all softkey values on the screen, with all 15 characters of each key displayed.										
KEY <i>n</i> , <i>stringexp</i>	<p>Sets function key <i>n</i></p> <p>Any one or all of the ten function keys may be assigned up to a 15 byte string by KEY <i>n</i>,<i>stringexp</i>. When the key is pressed, the associated string will be generated.</p> <p>Initially, the softkeys default to the following values:</p> <table> <tr> <td>F1 - LIST</td><td>F2 - RUN CR</td></tr> <tr> <td>F3 - LOAD"</td><td>F4 - SAVE"</td></tr> <tr> <td>F5 - CONT CR</td><td>F6 - ,"LPT1:" CR</td></tr> <tr> <td>F7 - TRON CR</td><td>F8 - TROFF CR</td></tr> <tr> <td>F9 - KEY space</td><td>F10 - SCREEN 0,0,0 CR</td></tr> </table>	F1 - LIST	F2 - RUN CR	F3 - LOAD"	F4 - SAVE"	F5 - CONT CR	F6 - ,"LPT1:" CR	F7 - TRON CR	F8 - TROFF CR	F9 - KEY space	F10 - SCREEN 0,0,0 CR
F1 - LIST	F2 - RUN CR										
F3 - LOAD"	F4 - SAVE"										
F5 - CONT CR	F6 - ,"LPT1:" CR										
F7 - TRON CR	F8 - TROFF CR										
F9 - KEY space	F10 - SCREEN 0,0,0 CR										

COMMANDS, STATEMENTS AND FUNCTIONS

Remarks

1. If the function key number is not in the range 1 to 10, an "Illegal function call" error is returned. The previous key string expression is retained.
2. The key assignment string may be 1 to 15 characters in length. If the string is longer than 15 characters, the first 15 characters are assigned.
3. Assigning a null string (string of length 0) to a softkey disables the function key as a softkey.
4. When a softkey is assigned, the INKEY\$ function returns one character of the softkey string per invocation.

Example

50 KEY ON	'Display the softkeys on the bottom line.
60 KEY OFF	'Erase softkey display.
70 KEY 1,"MENU"+CHR\$(13)	'Assigns the string "MENU" CR to soft key 1. Such assignments might be used for rapid data entry.
80 KEY 2,""	'Disables softkey 2 as a soft key.

The following routine initializes the first 5 softkeys:

```
1 KEY OFF 'Turn off key displays during init.
10 DATA "EDIT", "LEFT", "SYSTEM", "PRINT", "LPRINT"
20 FOR I = 1 TO 5:READ SOFTKEYS$(I)
30 KEY I, SOFTKEYS$(I)
40 NEXT I
50 KEY ON 'now display new softkeys.
```

Defining Keys 15-20

Defining keys 15 to 20 allows you to trap any Ctrl-Key, Shift-Key, or Super-Shift (**ALT**)-Key. These keys are defined by the statement:

KEY *n* , **CHR\$(shift) + CHR\$(scan-code)**

Where

SYNTAX ELEMENT	MEANING										
<i>n</i>	Is an integer expression in the range 15 to 20										
<i>shift</i>	<p>Is a numeric value corresponding to the following hex values:</p> <table><tr><td>CAPS LOCK</td><td>&H40 (Caps Lock is active)</td></tr><tr><td>NUM LOCK</td><td>&H20 (Num Lock is active - Keyboard 2 only)</td></tr><tr><td>ALT</td><td>&H08 (Alt Key is pressed)</td></tr><tr><td>Right SHIFT</td><td>&H01</td></tr><tr><td>Left SHIFT</td><td>&H02</td></tr></table> <p>Both the left and right SHIFT keys can be used, where values of &H01, &H02 or &H03 (the sum of hex 01 and hex 02) denote a SHIFT key.</p> <p>It is also possible to add multiple shift states, such as CTRL and ALT keys together, by adding the associated shift state values.</p>	CAPS LOCK	&H40 (Caps Lock is active)	NUM LOCK	&H20 (Num Lock is active - Keyboard 2 only)	ALT	&H08 (Alt Key is pressed)	Right SHIFT	&H01	Left SHIFT	&H02
CAPS LOCK	&H40 (Caps Lock is active)										
NUM LOCK	&H20 (Num Lock is active - Keyboard 2 only)										
ALT	&H08 (Alt Key is pressed)										
Right SHIFT	&H01										
Left SHIFT	&H02										
<i>scan-code</i>	Is a decimal number in the range 1 to 83 on Keyboard 1 and 1 to 103 on Keyboard 2. It represents the scan code (in decimal) of the key to be trapped.										

COMMANDS, STATEMENTS AND FUNCTIONS

Remarks

Trapped keys are processed in the following order:

1. **CTRL PRT SC**. Note that **CTRL PRT SC**, even if defined as a trappable key, will still produce a printed copy of all text sent to the screen.
2. The function keys **F1** to **F10**, and the cursor direction key. Defining a function key or cursor movement key as a user defined key trap will have no effect as they are considered pre-defined.
3. The user defined keys are examined (15-20).

Any key that is trapped is not passed to GW-BASIC, i.e., it does not go into the keyboard buffer. This applies to any key, including **CTRL BREAK** or **CTRL ALT DEL**. This makes it possible to prevent GW-BASIC users from accidentally interrupting a program or rebooting the system.

Examples

See the **ON KEY(n) GOSUB** statement.

KEY(n) Statements

KEY(n) ON enables, **KEY(n) OFF** disables, and **KEY(n) STOP** suspends event trapping of the specified key. The **KEY(n)** statement is usually used in a program.

KEY(n) {ON | OFF | STOP}

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	<p>Is an integer expression from 1 to 20. It specifies the number of a function key (if it is from 1 to 10), or a cursor direction key (11 to 14), or a user defined key (15 to 20). The cursor direction keys are:</p> <p>cursor up (11), cursor left (12), cursor right (13), and cursor down (14).</p> <p>User defined keys are specified with the formula:</p> <p>KEY <i>n</i>, CHR\$(<i>shift</i>) + CHR\$(<i>scan-code</i>)</p> <p>See the KEY statement.</p>

To Enable or Disable KEY(*n*) Trapping

IF...	THEN...
a KEY(<i>n</i>) ON is executed	KEY(<i>n</i>) trapping is enabled.
a KEY(<i>n</i>) OFF is executed	KEY(<i>n</i>) trapping is disabled.

COMMANDS, STATEMENTS AND FUNCTIONS

IF...	THEN...
a KEY(<i>n</i>) STOP is executed	KEY(<i>n</i>) trapping is suspended. If KEY(<i>n</i>) is pressed the event is remembered, and the trapping routine will be activated as soon as a KEY(<i>n</i>) ON is executed.
an ON KEY(<i>n</i>) GOSUB is executed	KEY(<i>n</i>) trapping is suspended.
an error trap takes place	all trapping is automatically disabled.

Example

```
10 KEY 4, "SCREEN 0,0,0" 'assign softkey 4
20 KEY (4) ON 'enables KEY trapping
.
.
100 ON KEY (4) GOSUB 1000
.
.
Key 4 pressed
.
.
1000 REM KEY (4) Trap Routine
```

KILL Command

Deletes a disk file.

KILL {*filespec*|*pathname*}

Where

SYNTAX ELEMENT	MEANING
<i>filespec</i> or <i>pathname</i>	Is a string expression which specifies the file to be deleted. The filename must include the extension if one exists.

Characteristics

KILL checks to see if the file is open, and if so will give a "File already open" error. KILL, like OPEN, cannot distinguish a file in another directory from one you may have open. It is possible to get an unexpected "File already open" error under these circumstances.

KILL can only be used to delete a file. You must use the RMDIR command to remove a directory.

Warning

The *filename* may contain question marks (?) or asterisks (*) used as wildcards. Be extremely careful when using wildcards with this command.

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
200 KILL "A:DATA1.DAT"  
300 KILL "C:DIR1\DIR2\PROG2.BAS"
```

Note that the filename must include the extension, if one exists. GW-BASIC does not supply the extension .BAS for the KILL command.

LCOPY Command

Prints the contents of the screen (text and or graphics). To print graphics you must have a graphics printer and have executed the MS-DOS GRAPHICS command before having entered GW-BASIC.

LCOPY [*n*]

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is a dummy argument. Any value may be supplied. This parameter is allowed only for compatibility with other BASICs, where it may specify to copy either text or graphics.

LEFT\$ Function

Returns a substring extracting a number of characters to the left of a given string, as specified by the *length* parameter.

LEFT\$(*string* , *length*)

Where

SYNTAX ELEMENT	MEANING
<i>string</i>	Is a string expression whose value is the string from which the substring is to be returned
<i>length</i>	Is an integer expression (from 0 to 255) which specifies the number of the characters to be returned.

Characteristics

If *length* is greater than LEN(*string*), the entire original string will be returned. If *length* = 0, the null string (length zero) will be returned.

Refer to the MID\$ and RIGHT\$ functions.

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
LIST
10 A$="GW-BASIC"
20 B$=LEFT$(A$,6)
30 PRINT B$
RUN
GW-BAS
Ok
```

LEN Function

Returns the length of a given string.

LEN(*stringexp*)

Where

SYNTAX ELEMENT	MEANING
<i>stringexp</i>	Is any string expression, whose length will be returned

Characteristics

Unprintable characters and blanks are counted in the number of characters. If the argument *stringexp* is a null string, LEN returns zero.

Example

```
10 X$ = "PORTLAND, OREGON"  
20 PRINT LEN(X$)  
RUN  
16  
Ok
```

LET Statement

Assigns a value to a variable.

[LET] *variable* = *expression*

Where

SYNTAX ELEMENT	MEANING
<i>variable</i>	Is a numeric or string variable which will receive the value of the expression.
<i>expression</i>	Is the expression to be evaluated and assigned to the variable on the left side of the equal sign

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

The type of the *expression* (numeric or string) must match the type of the *variable*; if not, a "Type Mismatch" error occurs. However in numeric assignments the type of the *expression* (integer, single precision or double precision) may be different from the type of the destination *variable*. In this case GW-BASIC converts the *expression* value to the type of the *variable*. Rounding or overflow may occur in this conversion.

The word LET is optional. The equal sign is sufficient when assigning an *expression* to a *variable* name.

Examples

```
110 LET D = 12
120 LET E = 12 ^ 2
130 LET F = 12 ^ 4
140 LET SUM = D + E + F
150 A(I) = 300
160 A$(K) = "ABC"
```

·
·
·

Note that the word LET is optional (see the statements in lines 150 and 160 above).

LINE Statement

Draws either a line or a rectangle, or a filled rectangle (Graphics Mode only).

LINE [[**STEP**] (*x1* , *y1*)] - [**STEP**] (*x2* , *y2*) [, [*color*] [, **B [F]**] [, *style*]]

Where

SYNTAX ELEMENT	MEANING
$(x1,y1)-(x2,y2)$	Represent absolute coordinates, or relative coordinates. If $(x1,y1)$ is omitted the last referenced point is assumed.
<i>color</i>	Is the color number specifying the color in which the line or rectangle will be drawn (in the range 0 to 3). In Medium Resolution, <i>color</i> chooses a color from the active palette; in High and Super Resolution, a value for <i>color</i> of 0 or 2 selects black and a value of 1 or 3 selects white. If <i>color</i> is omitted the color used will correspond to the value given for the <i>gforeground</i> parameter in the most recently executed COLOR statement; if this parameter has not been specified, 3 is assumed for Medium Resolution and 1 for High and Super Resolution.
B	Specifies the drawing of a rectangle
F	Specifies the drawing of a filled rectangle
<i>style</i>	Is a 16-bit parameter, normally given in hexadecimal format, that specifies the "style" of the line. It has no effect if BF has been specified.

Characteristics

The following example draws a line from the last point referenced to the point specified $(x2,y2)$. Since no color is specified, the default color is the foreground color.

LINE $-(x2,y2)$

COMMANDS, STATEMENTS AND FUNCTIONS

The examples below specify start and end points in absolute coordinates.

LINE (10,10)-(319,199)

draws a diagonal line down the screen

LINE (10,100)-(319,100)

draws a horizontal line across the screen

You can specify the color number in which the line is drawn:

LINE (15,15)-(25,25),2

draws a line in color 2 of the active palette.

The B parameter is used to draw a rectangle, where the points (x1,y1) and (x2,y2) represent the opposite corners. In the following example, no color number is specified:

LINE (10,10)-(100,100),,B

draws a box

Color may be included as follows:

LINE (10,10)-(200,200),2,BF

filled box color 2

The B parameter facilitates the drawing of rectangles, which would otherwise require the following lengthy programming format:

LINE (x1,y1)-(x2,y1)

LINE (x1,y1)-(x1,y2)

LINE (x2,y1)-(x2,y2)

LINE (x1,y2)-(x2,y2)

The BF options fills the interior of the rectangle with the selected *color*.

If the STEP option is specified for the second pair of coordinates, they are taken as relative to the first pair of coordinates. For example:

LINE(50,50) -STEP(15,-13)

draws a line from (50,50) to (65,37).

Line Styling

The *style* parameter can also be specified in the LINE statement. It is a 16-bit integer value, used to specify a mask for the style of a line. *style* can be used for drawing both lines and rectangles.

Before LINE plots a pixel on the screen, the value of the current bit in *style* is used. If the bit equals 0, the color of the pixel is unchanged; if the bit equals 1, the pixel is plotted with the color specified. For each successive pixel the successive bit-position in *style* is used. When the whole mask has been scanned, scanning restarts at the beginning of the mask, until the entire line is drawn.

Since a bit equal to 0 in *style* does not modify the color of the pixel on the screen, it can be useful to draw a line of a specific color, before overlaying it with a "styled" line, in order to have the correct background color for the line you wish to draw.

Example

```
LINE (0,0)-(160,100),3,,&HFF00
```

Draws a dashed line from the upper left hand corner to the screen center, assuming a screen 320 pixels wide by 200 pixels high.

LINE INPUT Statement

Inputs an entire line (up to 254 characters) to a string variable, without the use of delimiters. A LINE INPUT statement is only used in a program.

```
LINE INPUT[;] [prompt ; ]stringvar
```

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
<i>prompt</i>	Is a string constant (enclosed in a pair of quotation marks) that is displayed on the screen before input is accepted. A question mark is not displayed unless it is part of the "prompt string".
<i>stringvar</i>	Is the name of a string variable to which the line will be assigned.

Characteristics

All input from the end of *prompt* to the **CR** is assigned to *stringvar*. Trailing blanks are ignored. If a **LF CR** is encountered, both characters are echoed, but the **CR** is ignored, the **LF** is put into *stringvar*, and data input continues.

If **LINE INPUT** is immediately followed by a semicolon, then the **CR** typed to end the input line does not echo a **CR LF** sequence on the screen.

A **LINE INPUT** statement may be aborted by typing **CTRL BREAK**. **GW-BASIC** will return to command level and type "Ok". Typing **CONT** resumes execution at the **LINE INPUT**.

You may use all the **GW-BASIC** screen editor features in responding to **INPUT** and **LINE INPUT** statements.

Example

See **LINE INPUT #** statement.

LINE INPUT # Statement

Reads an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable. LINE INPUT # is usually used in a program.

LINE INPUT # *filenum* , *stringvar*

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is the number under which the file was OPENed
<i>stringvar</i>	Is the string variable to which the line will be assigned

Characteristics

LINE INPUT # reads all characters in the sequential file up to a CR . It then skips over the CR LF sequence. The next LINE INPUT # reads all characters up to the next CR . (If a line LF CR sequence is encountered, it is preserved.)

LINE INPUT # is especially useful if each line of a data file has been broken into fields, or if a GW-BASIC program saved in ASCII format is being read as data by another program. (See "SAVE," in this chapter.)

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES 234,4 MEMPHIS
LINDA JONES 234,4 MEMPHIS
Ok
```

LIST Command

Lists the current program to the screen or a specified file or device.
LIST is usually used in immediate mode.

LIST [*linenum1*][- [*linenum2*]][, {*device*|*filespec*|*pathname*}]

Where

SYNTAX ELEMENT	MEANING
<i>linenum1</i> and <i>linenum2</i>	Are the line numbers of the first and the last line to be listed. You may use a period (.) for either line number to indicate the current line.
<i>device</i> or <i>filespec</i> or <i>pathname</i>	Is a string expression for the device specification (or for the file specification).

Characteristics

If you omit the *device* or *filespec* or *pathname* the listing is directed to the screen. You can stop listings directed to either the screen or the printer by pressing **CTRL BREAK** at any time. You cannot interrupt listings directed to a file or device: in this case the listing will continue until the range is exhausted.

If you omit the line range, the complete program will be listed.

The syntax allows the following options:

- If only *linenum 1* - is given, that line and all higher numbered lines are listed.
- If only *linenum 2* - is given, all lines from the beginning of the program through the given line are listed.
- If both numbers are specified, the inclusive range is listed.

When you direct a listing to a disk file, the program is saved in ASCII format, thus you may later use this file with MERGE.

COMMANDS, STATEMENTS AND FUNCTIONS

Examples

LIST , LPT1:

List the program to the Line Printer.

LIST 10-90

List lines 10 through 90 to the Screen.

LIST 10- , SCRN:

List lines 10 through last to the Screen.

LLIST Command

Lists the current program on the printer. LLIST is usually used in immediate mode.

LLIST [*linenum1*][- [*linenum2*]]

Characteristics

For an explanation of the meaning of the parameters see the LIST command.

LLIST assumes a 132-character-wide printer.

GW-BASIC always returns to command level after an LLIST is executed.

Examples

LLIST

Lists a complete program.

LLIST 50

Lists line 50.

LLIST 20-40

Lists lines 20-40

LLIST-150

Lists from the first program line to line 150

LLIST 90-

Lists all lines from 90 to end of program.



LOAD Command

Loads a program into memory from a specified drive. You can run the program, if you specify the option R.

LOAD {*filespec*|*pathname*} [,R]

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
<i>filespec (or pathname)</i>	Is a string expression which specifies the file to be loaded. If you do not specify a drive letter, then the MS-DOS default drive is assumed.
R	Represents RUN . It is optional and when specified will cause the loaded program to begin execution from the first statement. In this case all open data files are kept open.

Characteristics

LOAD deletes all variables and program lines currently residing in memory and closes all open data files before it loads the specified program. However, if option R is specified, all open data files are kept open and the program is run after it is loaded.

Note that:

RUN filespec is equivalent to **LOAD filespec ,R**.

and that:

RUN pathname is equivalent to **LOAD pathname ,R**.

Example

LOAD "STRTRK",R

Loads and runs the program STRTRK.BAS

LOAD "B:MYPROG"

Loads the program MYPROG.BAS from the disk in drive B, but does not run the program.

LOC Function

Returns the current position in the file. LOC is usually used in a program.

LOC(*filenum*)

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is the number under which the file was OPENed

Characteristics

For random disk files, LOC returns the record number just read or written from a GET or PUT statement.

For sequential files, LOC returns the current byte position in the file, divided by 128.

When a file is opened for APPEND or OUTPUT, LOC returns the size of the file in (bytes/128).

COMMANDS, STATEMENTS AND FUNCTIONS

For communications files LOC is used to determine if there are any characters in the input queue waiting to be read. The input queue can hold more than 255 characters (determined by the /C: switch). If there are more than 255 characters in the queue, LOC returns 255. Since strings are limited to 255 characters, this practical limit removes the need to test for string size before reading data into them. If fewer than 255 characters remain in the queue, the value returned by LOC depends on whether the device was opened in ASCII or binary mode.

In either mode, LOC will return the number of characters that can be read from the device. However, in ASCII mode, the low level routines stop queueing characters as soon as end-of-file is received. The character which indicates the end-of-file itself is not queued and cannot be read. An attempt to read the end-of-file will result in an "Input past end" error.

Example

```
100 IF LOC(2) > 100 THEN STOP
```

LOCATE (Text) Statement

Moves the cursor to the specified position on the active page. LOCATE may also turn the user cursor on and off and define the size of either the user cursor, or both the user and overwrite cursors.

LOCATE [*row*][, [*column*][, [*cursor*][, [*start*][, *stop*]]]]

Where

SYNTAX ELEMENT	MEANING
<i>row</i>	Is the screen line number. A numeric expression returning an unsigned integer in the range 1 to 25.
<i>column</i>	Is the screen column number. A numeric expression returning an unsigned integer in the range 1 to 80, depending upon screen width.
<i>cursor</i>	Is a boolean value indicating whether the user cursor is visible or not. A zero value turns the user cursor off, a nonzero value (say 1) turns the cursor on.
<i>start</i>	Is a numeric expression whose integer value represents the cursor top (starting) scan line. If <i>start</i> is between 0 and 7 the size of both cursors (user and overwrite) is defined. If <i>start</i> is between 32 and 39 only the size of the user cursor is defined. Other values are invalid.
<i>stop</i>	Is a numeric expression whose integer value represents the bottom (stop) cursor scan line. It has the same range of values as <i>start</i> , but must be greater than or equal to <i>start</i> ; if it is equal the cursor will be shown as a single scanline. If this parameter is omitted, and <i>start</i> is given, <i>stop</i> defaults to the value of <i>start</i> .

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

There are three cursors in GW-BASIC:

- The insert-mode cursor, which appears when you are inserting characters into a GW-BASIC line.
- The overwrite cursor, which appears when you are entering a GW-BASIC line and when entering data after the execution of an INPUT statement.
- The user cursor, which can be made to appear during program execution (but not during the execution of an INPUT statement). It can be particularly useful to provide the user cursor for entering data after a INKEY\$ or INPUT\$ function, as the overwrite cursor does not appear.

The overwrite cursor is the most commonly used cursor, it is initialized as a two scanline underline; the insert-mode cursor is initialized as 4 scanlines in height; the user cursor is initially disabled (however its size is initialized to a full-size block). The size of the insert-mode cursor is fixed; the size of the overwrite and user cursors can be programmed.

If any of the parameters used with LOCATE are omitted, the current value is assumed.

Placing Characters on Line 25

Normally, GW-BASIC will not print to line 25 because of the soft key display. This can be turned off, however, using KEY OFF; you can then use LOCATE 25,1: PRINT... to display characters on line 25. PRINT statements on line 25 must end with a semicolon, otherwise the screen may scroll under certain circumstances. The following sequence of statements will result in a 25-line scrolling display, without any function key line:

```
KEY OFF  
VIEW PRINT
```

Examples

100 LOCATE 1,1

Moves the cursor to the home position in the upper left-hand corner.

200 LOCATE ,,1

Makes the user cursor visible; its position remains unchanged.

300 LOCATE ,,0

Turns the user cursor off.

400 LOCATE 6,1,1,0,7

Moves the cursor to line 6, column 1. Makes the user cursor visible, covering the entire character cell, starting at scan line 0 and ending on scan line 7.

500 LOCATE ,,1,13

Makes the user cursor visible; its position remains unchanged, the shape of the user and overwrite cursor will be a thin horizontal line at the bottom of the character cell.

600 LOCATE ,,1,45

Makes the user cursor visible, its position remains unchanged, its shape will be a thin horizontal line at the bottom of the character cell.

Possible Errors

Any values entered outside of the ranges indicated will result in an "illegal function call" error. Previous values are retained.

LOCATE (Graphics) Statement

Moves the cursor to the specified position. LOCATE may turn the user cursor on and off and define the shape or shapes (simultaneously) of the user and overwrite mode cursor. LOCATE may also specify the blinkrate of the user cursor.

Syntax 1

LOCATE [*row*][, [*column*][, [*rate*][, [*start*][, *stop*]]]

Syntax 2

LOCATE [*row*][, [*column*][, [*rate*][, *line* , *map*]]]

Where

The value of the fourth parameter determines whether Syntax 1 or 2 applies.

SYNTAX ELEMENT	MEANING
<i>row</i>	Is the screen line number. A numeric expression returning an unsigned integer in the range 1 to 25.
<i>column</i>	Is the screen column number. A numeric expression returning an unsigned integer in the range 1 to 80, depending upon screen width.

SYNTAX ELEMENT	MEANING
<i>rate</i>	<p>Is an integer expression in the range 0 to 10, which determines the state of the user cursor.</p> <p>0 Turn the user cursor off 1 Turn the user cursor on 2...10 Blink the user cursor with a period of <i>rate</i> units of 1/18.75 seconds</p>
<i>start</i>	<p>Is the cursor starting scanline. It must be an integer expression in the range 0 to 15 (or 32 to 47) in Super Resolution and between 0 and 7 (or 32 and 39) in Medium and High Resolution. If <i>start</i> is in the range 0 to 15 the shapes of both the user and the overwrite cursor will be programmed. If <i>start</i> is in the range 32 to 47 it is taken modulo 16, and only the user cursor shape will be programmed.</p>
<i>stop</i>	<p>Is the cursor stop scanline. It must be an integer expression with values in the same range as <i>start</i>, but it must be greater than or equal to <i>start</i>: if it is equal the cursor will be a single scanline.</p>
<i>line</i>	<p>If the value of <i>line</i> is between 50 and 50 + M, byte number <i>line</i> - 50 of the overwrite cursor bitmap is set to <i>map</i>. If the value is between 100 and 100 + M, then byte number <i>line</i> - 100 of the user cursor bitmap is set to <i>map</i>. The value of M is 7 for High-Resolution, and 15 for Medium and Super-Resolution.</p>

COMMANDS, STATEMENTS AND FUNCTIONS

SYNTAX ELEMENT	MEANING
<i>map</i>	Replaces the contents of the byte specified using <i>line</i> . The cursor bitmap is a byte array which is XOR'd with the pixels on the screen. In Medium Resolution, each scanline of the cursor is represented by two bytes; each pixels in the cursor is associated with two consecutive bits which specify the color of the pixel. In High and Super Resolution there is one byte per scanline. All the Graphics modes (Medium, High and Super) store a bitmap for each type of cursor (overwrite, insert and user) which are restored if you change from one screen mode to another.

Characteristics

GW-BASIC includes a blinking cursor for graphics. The maximum height of this cursor equals the height of the character cell; therefore, it equals 8 scanlines in Medium and High Resolution and 16 scanlines in Super Resolution.

Cursor scanlines are numbered starting with 0 for the top scanline.

You can define three different cursors in each of the graphics mode; as well as in text mode.

The insert-mode cursor will always be a rapidly-blinking small triangle at the lower left of the character cell. The overwrite-mode cursor is initially an underline, which blinks more slowly. The user cursor is initially disabled. The shape arrays for the user and overwrite cursors are initially loaded with 0FFH bytes in order that they can easily be made into underline or block shapes. The shapes of the user and overwrite cursors are programmable. The blinkrate of the user cursor is programmable, but the blinkrates of the overwrite and insert cursors are fixed.

LOCATE,,0 will disable only the user cursor. Also, execution of any graphics statement (LINE, PSET, etc.) will disable the user cursor.

The overwrite cursor will always appear whenever an INPUT statement is executed and when you are entering or correcting a line of GW-BASIC. The insert cursor appears when you insert characters into a GW-BASIC line. Otherwise only the user cursor will appear.

Examples

```
10 LOCATE 5,1,4,5,7
```

Moves the cursor to line 5, column 1, turns the user cursor on with a blinkrate of 4/18.75 seconds and sets the height of the user and overwrite cursors to 3. The shape array of the cursor is initialized to 0FFH, (unless the user has changed the bitmap).

```
100 LOCATE ,,51,&H82  
110 LOCATE ,,103,&H01
```

These statements set bytes in the bitmaps for the overwrite and user cursor, respectively. Statement 100 sets byte 1 of the overwrite cursor to Hex 82; statement 110 sets byte 3 of the user cursor to Hex 01.

```
10 SCREEN 1
```

```
50 FOR I = 0 TO 15  
60 LOCATE ,, 50 + I, 0  
70 NEXT I
```

This example clears the bitmap of the overwrite cursor (in Medium Resolution).

COMMANDS, STATEMENTS AND FUNCTIONS

```
10 SCREEN 2
.
.
.
50 FOR I = 0 TO 7
60 LOCATE,,,50+I,0
80 NEXT I
```

This example clears the bitmap of the overwrite cursor (in High Resolution).

```
10 SCREEN 3
.
.
.
50 FOR I = 0 TO 15
60 LOCATE,,,50+I,0
70 NEXT I
```

This example clears the bitmap of the overwrite cursor (in Super Resolution)

```
10 SCREEN 1
20 LOCATE,,,0,7
```

LOCK Statement

The LOCK statement restricts access by other processes to all or part of an opened file. This statement is only of use if MS-NET is installed with MS-DOS release 3.1 or later.

LOCK [#] *filename* [, *recordnum1*] [**TO** *recordnum2*]

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is the file number of the opened file.
<i>recordnum1</i>	Is the number of the record at which locking begins. If this parameter is omitted record number 1 is assumed.
<i>recordnum2</i>	Is the number of the record at which locking terminates. If this parameter is omitted, only one record (<i>recordnum1</i>) is locked.

Characteristics

If the file has been opened for sequential input or output, the entire file is locked regardless of any range specified. If the file is opened in random mode, the range indicates which records are to be locked.

See also the UNLOCK and OPEN statements.

Note

If you try and use this statement with an MS-DOS release prior to 3.1 you will get an "Advanced Feature Error". If you are using MS-DOS 3.1 or later, but MS-NET is not installed, you will get a "Permission Denied" error. The MS-DOS command SHARE must also be given (either at the MS-DOS prompt or from an AUTOEXEC.BAT file). See the "MS-DOS User Guide" for a full description of procedures to be followed for networking.

The suggested usage of files on shared devices is for a LOCK to be executed on a file, or record range within a file, before an attempt is made to read or write to that file. It is also recommended that the file or range be UNLOCKed before the file is closed (failure to do so may cause problems for future access to that file in a network environment). The time in which files are locked should be kept to minimum.

COMMANDS, STATEMENTS AND FUNCTIONS

Examples

LOCK #1	'locks the whole of file #1
LOCK #1, 2	'locks record 2 of file #1
LOCK #1, TO 23	'locks records 1 to 23 of file #1
LOCK #1, 12 TO 24	'locks records 12 to 24 of file #1

Possible Errors

"Permission Denied": MS-NET non installed or access to file already restricted.

"Illegal Function Call": record range specified does not meet necessary criteria, or range/record length combination exceeds the legal limit for the size of a file.

LOF Function

Returns the length of the specified file in bytes.

LOF(*filename*)

Characteristics

For random and sequential files, LOF returns the size of the file in bytes. Note that, when a file is opened for APPEND or OUTPUT, LOF returns the size of the file divided by 128.

For communications files, LOF may be used to check if the input buffer is getting full as it returns the amount of free space in the input buffer. That is:

buffer-size - **LOF(filename)**

where *buffer-size* is the size of the communications buffer. It defaults to 256 bytes, but may be changed with the "/C:" option in the GWBASIC command line.

Example

```
10 OPEN "B:MYFILE" AS #2
20 GET #2,LOF(1)/128
```

The above statements will get the last record of the file MYFILE (residing on the diskette inserted in drive B) assuming that the file was created with a record length of 128 bytes.

Example

```
100 IF REC*RECSIZ>LOF(1) THEN PRINT "INVALID ENTRY"
```

In this example, the variables REC and RECSIZE contain the record number and record length, respectively. The calculation determines whether the specified record is beyond the end-of-file.

LOG Function

Returns the natural logarithm of a positive argument.

LOG(*numexp*)

Characteristics

The logarithm is calculated in single precision, unless the /D option is supplied in the GWBASIC command line.

If *numexp* is negative or zero an "Illegal function call" error is returned.

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
PRINT LOG(45/7)
1.860752
Ok
```

LPOS Function

Returns the current position of the print head within the printer buffer.
LPOS is usually used in a program.

LPOS(*printer*)

Where

SYNTAX ELEMENT	MEANING
<i>printer</i>	Is an integer expression whose value (1, 2, or 3) indicates which printer is to be tested (LPT1:, LPT2:, or LPT3:).

Remarks

LPOS does not necessarily give the physical position of the print head.

Example

```
150 IF LPOS(1)<60 THEN LPRINT CHR$(13)
```


LPRINT and LPRINT USING Statement

LPRINT prints data in standard format on the printer. LPRINT USING prints data in a user-defined format on the printer.

Syntax 1

LPRINT [*list-of-expressions*][, | ;]

Syntax 2

LPRINT USING *format-string* ; *list-of-expressions*[, | ;]

Where

SYNTAX ELEMENT	MEANING
<i>list-of-expressions</i>	May include numeric and/or string expressions separated by commas or semicolons.
<i>format-string</i>	Is a string expression (usually a string constant or variable) that is composed of special formatting characters.

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

These statements are the same as PRINT and PRINT USING, but the output goes to the printer. See PRINT and PRINT USING in this chapter. LPRINT assumes a 132 character-width printer.

Example

```
10 A$ = "For July..."
20 X = .491
30 LPRINT "Results", A$,
40 LPRINT X
Ok
RUN
Results          For July...      .491
```

The result appears on the line printer.

LSET and RSET Statements

LSET stores a string value in a random buffer field left justified, or left justifies a string value in a string variable.

RSET stores a string value in a random buffer field right justified, or right justifies a string value in a string variable.

LSET and RSET are usually used in a program.

Syntax 1

LSET *stringvar* = *stringexp*

Syntax 2

RSET *stringvar* = *stringexp*

Where

SYNTAX ELEMENT	MEANING
<i>stringvar</i>	Represents a variable previously used in a FIELD statement.
<i>stringexp</i>	Represents the string to be left or right justified in a given field.

Characteristics

If *stringexp* requires fewer bytes than were FIELDed to *stringvar*, LSET left-justifies the string in the field, and RSET right-justifies the string (spaces are used to pad the extra positions). If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See also Chapter 4 for a complete description of random files.

Examples

```
150 LSET A$ = MKS$(AMT)
160 LSET D$ = MKI$(COUNT%)
```

COMMANDS, STATEMENTS AND FUNCTIONS

LSET or RSET may also be used with a nonfielded string variable to left-justify or right-justify a string in a given field. For example, the program lines:

```
110 A$ = SPACE$(20)
120 RSET A$ = N$
```

right-justify the string N\$ in a 20-character field. This can be most useful for formatting printed output.

MERGE Command

Merges a specified GW-BASIC disk file, previously saved in ASCII format, with the program currently resident in memory.

MERGE {*filespec* | *pathname*}

Where

SYNTAX ELEMENT	MEANING
<i>filespec</i> (or <i>pathname</i>)	Is a string expression which specifies a GW-BASIC source file, and optionally the drive on which it can be found. If the drive is omitted the MS-DOS default drive is assumed.

Characteristics

The MERGE command allows you to include a specified program saved (in ASCII format) on a disk, with the program in memory. MERGE is similar to LOAD, except that the program in memory is not erased before the disk program is transferred into memory. Instead, the disk program is merged into the resident program in such a way as to make a single program. In other words, program lines in the disk program will simply be inserted into the resident program in sequential order. If a line of the disk program and a line of the resident program have the same line number, the line of the disk program replaces that of the resident program in memory.

Example

```
MERGE "B:ROOT\S1\SUBRTN"
```



MID\$ Function and Statement

As a function, MID\$ returns a substring from a specified string. As a statement, MID\$ replaces a part of a string with another string.

Syntax 1: function

```
MID$( string , start[ , length] )
```

Syntax 2: statement

```
MID$( string , start[ , length] ) = substring
```

COMMANDS, STATEMENTS AND FUNCTIONS

Where: As a Function

SYNTAX ELEMENT	MEANING
<i>string</i>	Is a string expression whose value is the string from which the substring is to be returned
<i>start</i>	Is an integer expression whose value specifies the character position of the beginning of the returned string. It must be ≥ 1 .
<i>length</i>	Is an integer expression from 0 to 255 which represents the length of the returned string

Characteristics

The function returns a substring from a specified string, starting from a specified character position (*start*). The *length* of the returned substring can be specified. If *length* is omitted or if there are fewer than *length* characters to the right of the specified character position, all rightmost characters beginning with the specified character position are returned. If *length* is equal to zero, or if *start* is greater than LEN (*string*), then MID\$ returns a null string.

Also see LEFT\$ and RIGHT\$ functions.

Where: As a Statement

SYNTAX ELEMENT	MEANING
<i>string</i>	Is a string expression whose value is the string from which a substring is to be replaced.
<i>start</i>	Is an integer expression from 1 to 255, whose value specifies the character position where the replacement has to begin; <i>start</i> must be $\leq \text{LEN}(\textit{string})$.
<i>length</i>	Is an integer expression from 0 to 255. It represents the length of the returned string.
<i>substring</i>	Is a string expression which replaces the characters in <i>string</i> , beginning from <i>start</i> position.

Characteristics

The characters in *string*, beginning from *start* position, are replaced by the characters in *substring*. The optional *length* refers to the number of characters from *substring* that will be used in the replacement. If *length* is omitted, all of the characters of *substring* are used.

However, regardless of whether *length* is omitted or included, the replacement of characters never goes beyond the original length of *string*.

COMMANDS, STATEMENTS AND FUNCTIONS

Examples

The following example uses the MID\$ function:

```
LIST
10 A$ = "HELLO "
20 B$ = "JOSEPH JOHNNY JIMMY"
30 PRINT A$;MID$(B$,8,6)
RUN
HELLO JOHNNY
Ok
```

The following example uses the MID\$ statement:

```
LIST
10 A$ = "AVIGNON, FRANCE"
20 MID$(A$,10) = "ROUBAIX"
30 PRINT A$
RUN
AVIGNON, ROUBAI
Ok
```

Possible Errors

If either *start* or *length* is out of the specified range, an "Illegal function call" error will be returned.

MKDIR Command

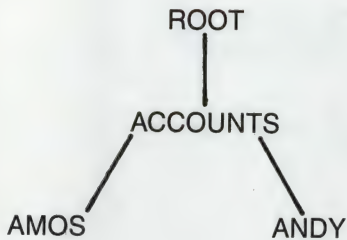
Permits the creation of a new directory on a specified disk.

MKDIR *pathname*

Where

SYNTAX ELEMENT	MEANING
<i>pathname</i>	Is a string expression specifying the name of the directory to be created.

Examples



With reference to the above tree-structure, the following statements will create sub-directories.

If the current directory is the root, to create a sub-directory SALES from the root on the current drive, enter:

MKDIR "SALES"

To create a sub-directory called FRED under the directory SALES, enter:

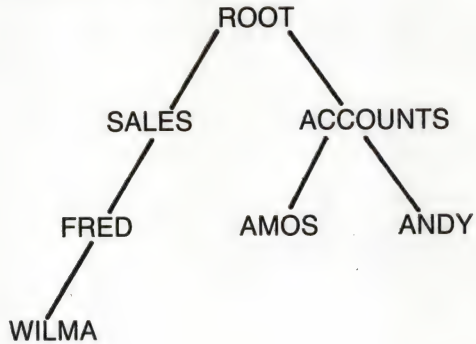
MKDIR "SALES\FRED"

To create a sub-directory called WILMA under the directory FRED, enter:

COMMANDS, STATEMENTS AND FUNCTIONS

MKDIR "SALES\FRED\WILMA"

The resulting structure will be:



Possible Errors

"Bad file name"

"Path/File Access error"

MKI\$,MK\$\$,MKD\$ Functions

Change numeric values to string type values.

Syntax 1

MKI\$(*integer-expression*)

Syntax 2

MKS\$(*single-precision-expression*)

Syntax 3

MKD\$(*double-precision-expression*)

Characteristics

Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI\$ converts an integer to a 2-byte string, MKS\$ converts a single-precision number to a 4-byte string. MKD\$ converts a double-precision number to an 8-byte string.

See also "CVI, CVS, CVD Functions".

Example

```
90 AMT=(K+T)
100 FIELD #1,8 AS D$,20 AS N$
110 LSET D$=MKS$ (AMT)
120 LSET N$=A$
130 PUT #1
```

NAME Command

Changes the name of a disk file.

Syntax 1

NAME *{filespec|pathname}* **AS** *filename*

Syntax 2

NAME *{filespec|pathname}* **AS** *new-pathname*

Where

SYNTAX ELEMENT	MEANING
<i>filespec</i> (or <i>pathname</i>)	Is a string expression which specifies the file to be renamed. The file must exist on the specified drive. If the drive is not specified the MS-DOS default drive is assumed. The file extension does not default to .BAS.
<i>filename</i>	Is the new name of a file to replace the old filename. The new name should not exist for another file.
<i>new-pathname</i>	Is a string expression which specifies the new directory of the file, when NAME is used to move a file from one directory to another.

Characteristics

After a NAME command, the file exists in the same area on the same disk, with the new name. Also the area allocated to the file will not be changed. A file may not be renamed with a new drive designation. If this is attempted, a "Rename across disks" error will be generated. The file must be closed before the NAME command is executed.

NAME may also be used to move a file from one directory to another. For example:

```
NAME "\X\CLIENTS" AS "\XYZ\P\CLIENTS"
```

Example

Ok

```
NAME "B:GRAPH.BAS" AS "GRAPH1.BAS"
```

Ok

In this example, the file that was formerly named GRAPH.BAS on the diskette in drive B will now be named GRAPH1.BAS.

Possible Errors

"File not found"

"File already exists"

"Bad file name"

"Too many files"

"Rename across disks"

NEW Command

Deletes the current program and clears all variables, allowing you to enter a new program.

NEW

Characteristics

NEW is entered at command level to clear memory before entering a new program. GW-BASIC always returns to command level after a NEW command is executed. NEW closes all data files and switches off the trace flag in the same way as TROFF.

OCT\$ Function

Returns a string which is the octal value of the decimal argument.

OCT\$(*numexp*)

Where

SYNTAX ELEMENT	MEANING
<i>numexp</i>	Is a numeric expression from -32768 to 65535, which is rounded to the nearest integer before OCT\$ is evaluated.

Remarks

When *numexp* is negative, the two's complement form is used.

Example

```
PRINT OCT$(24)
30
Ok
```

See the HEX\$ function in this chapter for details on hexadecimal conversion.

ON COM(*n*) GOSUB Statement

Specifies the first line number of a subroutine to be activated as soon as characters arrive in the communications buffer.

The ON COM(*n*) GOSUB statement is only used in a program.

ON COM(*n*) GOSUB *linenum*

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is the number of the communications channel (1 or 2).
<i>linenum</i>	Is the first line number of the trap routine. A line number of 0 disables the communications event trap.

To Enable or Disable COM Trapping

IF...	THEN...
a COM(<i>n</i>) ON is executed	COM(<i>n</i>) trapping will be enabled.
a COM(<i>n</i>) OFF is executed	COM(<i>n</i>) trapping will be disabled.
a COM(<i>n</i>) STOP is executed	COM(<i>n</i>) trapping will be suspended i.e. should any characters be received, the GOSUB is not performed, but will be performed as soon as a COM(<i>n</i>) ON statement is executed.
an ON COM(<i>n</i>) GOSUB is executed	COM(<i>n</i>) trapping will be suspended.
an error trap occurs	all trapping will be disabled (including ERROR, trapping).

To Activate a COM(*n*) Trapping Routine

IF...	THEN...
COM(<i>n</i>) trapping is enabled, and any characters have come into the COM buffer <i>n</i>	<p>the ON COM(<i>n</i>) GOSUB <i>linenum</i> will be executed and the corresponding routine activated. To avoid recursive traps a COM(<i>n</i>) STOP is automatically executed, when the trap occurs.</p> <p>A RETURN from the trap routine automatically performs a COM(<i>n</i>) ON (unless a COM(<i>n</i>) OFF was performed within the trap routine)</p> <p>The RETURN <i>linenum</i> form may also be used. Use this form with care because any other active GOSUB, WHILEs or FORs will remain active, and errors (such as "FOR without NEXT") may result.</p>

Typically, the COM trap routine will read an entire message from the COM port before returning. The COM trap should not be used for single character messages since, at high baud rates, the overhead of trapping and reading for each individual character may cause the COM interrupt buffer to overflow.

Example

This example sets up a trap routine for the second communications channel at line 1000.

```
100 ON COM(2) GOSUB 1000
110 COM(2) ON
```

COMMANDS, STATEMENTS AND FUNCTIONS

1000 REM COM activity

1050 RETURN 200

ON ERROR GOTO Statement

Enables error trapping and specifies the first line number of a subroutine to be executed if an error occurs.

The ON ERROR GOTO statement is only used in a program.

ON ERROR GOTO *linenum*

Where

SYNTAX ELEMENT	MEANING
<i>linenum</i>	Is the first line number of the error trapping routine

To Enable or Disable ERROR Trapping

IF...	THEN...
an ON ERROR GOTO <i>n</i> is executed	ERROR trapping will be enabled, if <i>n</i> is not zero.
an ON ERROR GOTO 0 is executed	ERROR trapping will be disabled. Subsequent errors will display the associated error message and halt execution.

To Activate an ERROR Trapping Routine

IF...	THEN...
ERROR trapping is enabled and a GW-BASIC error (or a user defined error) is found	<p>the ON ERROR GOTO line will be executed and the corresponding routine activated. The ERL and ERR functions are usually used in IF...GOTO...ELSE or IF...THEN...ELSE statements to direct program flow within an error trapping routine.</p> <p>It is recommended that the error trapping routine execute an ON ERROR GOTO 0 if an error is found for which there is no recovery action (in this case the standard error message will be displayed and execution will stop). The RESUME statement will resume execution after the error handling routine has been entered (see the RESUME statement).</p>

COMMANDS, STATEMENTS AND FUNCTIONS

IF...	THEN...
a GW-BASIC error (or a user-defined error) is found, during the execution of an error trapping routine	the associated error message is displayed and execution terminates. Note: Once an error trap takes place, all trapping is automatically disabled.

Example

```
10 ON ERROR GOTO 100
20 INPUT R
30 IF R <= 0 THEN ERROR 300
.
.
.
100 IF ERR = 300 THEN PRINT "RADIUS NEGATIVE OR ZERO"
.
110 IF ERL = 30 THEN RESUME 20
120 ON ERROR GOTO 0
.
.
.
```


ON...GOSUB and ON...GOTO Statements

ON...GOSUB calls one of several specified subroutines, depending on the value of a specified expression. ON...GOTO branches to one of several specified line numbers, depending on the value of a specified expression.

The ON...GOSUB and ON...GOTO statements are usually used in a program.

Syntax 1

ON *numexp* **GOSUB** *linenum* [, *linenum*]...

Syntax 2

ON *numexp* **GOTO** *linenum* [, *linenum*]...

Where

SYNTAX ELEMENT	MEANING
<i>numexp</i>	Is a numeric expression (from 0 to 255) which determines which line number in the list will be used for branching. If <i>numexp</i> is not an integer, it will be rounded up to an integer.
<i>linenum</i>	Is the line number to which the branch will be made.

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of *numexp* is 1 the subroutine at the first line number in the list will be called, a value of 2 causes the subroutine at the second line number in the list to be called and so on. If the value of *numexp* is zero or greater than the number of items in the list (but less than or equal to 255), GW-BASIC continues with the next executable statement.

If the value of *numexp* is negative or greater than 255, an "Illegal function call" error occurs.

Example

```
100 ON L-1 GOTO 150,300,320,390
```

ON KEY(*n*) GOSUB Statement

Specifies the first line number of a subroutine to be executed when a specified key is pressed. The ON KEY(*n*) GOSUB statement is only used in a program.

ON KEY (*n*) GOSUB *linenum*

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is an integer from 1-20. It specifies the key to be trapped as follows: 1-10 function keys F1 to F10 , 11 Cursor Up, 12 Cursor Left, 13 Cursor Right, 14 Cursor Down, 15-20 Keys defined in the form: KEY <i>n</i> , CHR\$(<i>shift</i>) + CHR\$(<i>scan code</i>).
<i>linenum</i>	Is the first line number of the routine that is to be performed when the specified function or cursor direction key is pressed. A line number of 0 disables the event trap.

To Enable or Disable KEY(*n*) Trapping

IF...	THEN...
KEY(<i>n</i>) ON is executed	KEY(<i>n</i>) trapping will be enabled.
KEY(<i>n</i>) OFF is executed	KEY(<i>n</i>) trapping will be disabled.
KEY(<i>n</i>) STOP is executed	KEY(<i>n</i>) trapping will be suspended, i.e. should a specified key be pressed, the GOSUB is not performed, but it will be performed as soon as a KEY(<i>n</i>) ON is executed.

COMMANDS, STATEMENTS AND FUNCTIONS

IF...	THEN...
an ON KEY(<i>n</i>) GOSUB is executed	KEY(<i>n</i>) trapping will be suspended.
an error trap occurs	all trapping will be disabled.

To Activate a KEY(*n*) Trapping Routine

IF...	THEN...
KEY(<i>n</i>) trapping is enabled, and key <i>n</i> was pressed	<p>ON KEY(<i>n</i>) GOSUB is executed and the corresponding routine activated.</p> <p>To avoid recursive traps a KEY(<i>n</i>) STOP is automatically executed, when the trap occurs. A RETURN from the trap routine automatically performs a KEY(<i>n</i>) ON (unless a KEY(<i>n</i>) OFF was performed within the trap routine). The RETURN <i>linenum</i> form may also be used. Use this form with care, because any other active GOSUBs, WHILEs, or FORs will remain active, and errors may result.</p>

Remarks

When a key is trapped, the event is not memorized. Therefore, you cannot subsequently use the INPUT or INKEY\$ statements to find out which key caused the trap. So if you wish to assign different functions to particular keys, you must set up a different subroutine for each key, rather than assigning the various functions within a single subroutine.

Example

```
10 KEY 4,"SCREEN 0,0" 'assigns softkey 4
20 KEY(4) ON 'enables event trapping
.
.
.
70 ON KEY(4) GOSUB 200
.
.
.
key 4 pressed
.
.
.
200 'Subroutine for screen
.
.
.
250 RETURN
```

In the above, the normal function associated with function key 4 is overridden, and replaced with "SCREEN 0,0", which will be displayed whenever that key is pressed. The value may be reassigned and it will resume its standard function when the system is rebooted.

Example

```
100 KEY 15, CHR$(&H04) + CHR$(83)
105 REM ** Key 15 now is CTRL DEL **
110 KEY(15) ON
```

COMMANDS, STATEMENTS AND FUNCTIONS

```
.  
.  
1000 PRINT "If you want to stop processing for a break"  
1010 PRINT "press the CTRL key and DEL at the"  
1020 PRINT "same time."  
1030 ON KEY(15) GOSUB 3000.
```

The user presses CTRL DEL

```
.  
3000 REM ** Suspend processing loop.  
3010 CLOSE #1  
3020 RESET  
3030 CLS  
3035 PRINT "Enter CONT to continue."  
3040 STOP  
3050 OPEN "A", #1, "ACCOUNTS.DAT"  
3060 RETURN
```

In the above, the CTRL DEL key is enabled to enter a subroutine which closes the files and stops program execution until the operator is ready to continue.

ON PLAY(*n*) GOSUB Statement

Specifies the first line number of a subroutine to be executed when the music buffer contains fewer than *n* notes. This permits continuous background music during program execution.

The ON PLAY(*n*) GOSUB statement is only used in a program.

ON PLAY(*n*) GOSUB *linenum*

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is an integer expression in the range 1 to 32. Values outside this range result in an "Illegal function call" error.
<i>linenum</i>	Is the first line number of the associated trap routine. A line number of 0 disables play trapping.

To Enable or Disable PLAY(*n*) Trapping

IF...	THEN...
a PLAY ON is executed	PLAY(<i>n</i>) trapping will be enabled.
a PLAY OFF is executed	PLAY(<i>n</i>) trapping will be disabled.
a PLAY STOP is executed	PLAY(<i>n</i>) trapping will be suspended, i.e. should the music buffer contain less than the specified number of notes, the GOSUB is not performed, but it will be performed as soon as a PLAY ON is executed.
an ON PLAY(<i>n</i>) GOSUB is executed	PLAY(<i>n</i>) trapping will be suspended.
an error trap occurs	all trapping will be disabled.

COMMANDS, STATEMENTS AND FUNCTIONS

To Activate a PLAY(*n*) Trapping Routine

IF...	THEN...
PLAY(<i>n</i>) trapping is enabled, and the background music buffer has gone from <i>n</i> to <i>n</i> -1 notes	ON PLAY(<i>n</i>) GOSUB line will be executed, and the corresponding routine activated. To avoid recursive traps, a PLAY STOP is automatically executed when the trap occurs. A RETURN from the trapping subroutine will automatically perform a PLAY(<i>n</i>) ON (unless a PLAY(<i>n</i>) OFF statement was performed within the trap routine). The RETURN <i>linenum</i> form may also be used. Use this form with care, because any other active GOSUB, WHILE, or FOR statements will remain active, and errors (such as "FOR without NEXT") may result.
the program is running, PLAY(<i>n</i>) trapping is enabled, and the background music buffer is empty	no PLAY(<i>n</i>) trapping routine will be executed

Remarks

1. A PLAY event trap is only effective when playing Background Music (PLAY "MB..."). PLAY event traps have no effect when running in Music Foreground (PLAY "MF...").
2. A PLAY event trap is ineffective if the Music Background buffer is already empty when a PLAY ON is executed.

3. Care should be taken in selecting values for n . If n is a large number, event traps will occur frequently enough to reduce program execution speed.

Example

```
10 PLAY ON
20 ON PLAY(8) GOSUB 1000
.
.
.
1000 'SUB PLAY(8) TRAP
.
.
.
1050 RETURN
```



ON TIMER (n) GOSUB Statement

Causes an event trap every n seconds.

The ON TIMER (n) GOSUB statement is only used in a program.

ON TIMER(n) GOSUB *linenum*

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is an integer expression in the range 1 through 86400 (1 second through 24 hours). Values outside this range will result in an "Illegal function call" error.
<i>linenum</i>	Is the entry point line number of the TIMER event trap subroutine.

Characteristics

The ON TIMER GOSUB statement will only be executed if a TIMER ON statement has been executed to enable event trapping. If event trapping is enabled, and if *linenum* in the ON TIMER GOSUB statement is not zero, GW-BASIC checks between statements to see if the time has been reached. If it has, a GOSUB will be performed to the specified line.

If a TIMER OFF statement has been executed the GOSUB is not performed and is not remembered.

If a TIMER STOP statement has been executed the GOSUB is not performed, but will be performed as soon as a TIMER ON statement is executed.

If an ON TIMER(*n*) GOSUB is performed, an automatic TIMER STOP is executed so that recursive traps cannot take place. The RETURN from the trapping subroutine will automatically perform a TIMER ON statement unless a TIMER OFF statement was performed inside the subroutine.

The RETURN *linenum* form of the RETURN statement may be used to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.

Example

To display the time of day on line 1 every minute:

```
10 ON TIMER (60) GOSUB 1000
20 TIMER ON

.
.
.
1000 OLDROW = CSRLIN ' save current row
1010 OLDCOL = POS(0) ' save current column
1020 LOCATE 1,1 : PRINT TIME$
1030 LOCATE OLDROW, OLDCOL 'restore row and column
1040 RETURN
```

OPEN Statement

Allows I/O to a file or device. OPEN is usually used in a program.

Syntax 1

```
OPEN {device|filespec|pathname} [ FOR mode1][access] AS [# ]
filenum [ LEN = record-length]
```

COMMANDS, STATEMENTS AND FUNCTIONS

Syntax 2

OPEN *mode2* , [#] *filenum* ,{*filespec*|*pathname*}[, *record-length*]

Where

SYNTAX ELEMENT	MEANING
<i>device</i>	Is a string expression which specifies the device to be opened
<i>filespec</i>	Is a string expression which specifies the file to be opened. It may optionally include a device.
<i>pathname</i>	Is a string expression which specifies the file to be opened. It may optionally include a device.
<i>mode1</i>	<p>Is a literal string not enclosed in quotation marks. It determines the initial file pointer position and the action to be taken if the file does not exist. The valid modes and actions taken are:</p> <p>INPUT: specifies sequential input mode. Positions the pointer to the beginning of an existing file. A "File not found" error is given if the file does not exist.</p> <p>OUTPUT: specifies sequential output mode. Positions the pointer to the beginning of the file. If the file does not exist, one is created.</p>

SYNTAX ELEMENT	MEANING
	<p>APPEND: specifies sequential output after the last record on the file. Positions the pointer to the end of the file. If the file does not exist, one is created.</p> <p>This mode is valid only for disk files.</p> <p>If the <i>FOR mode1</i> clause is omitted, the initial position is at the beginning of the file. If the file is not found, one is created in Random I/O mode (that is records may be read or written, at will, at any position within the file).</p>
access	<p>This parameter is used with systems that have MS-NET installed (MS-DOS releases 3.1 or later) and it specifies the access rights associated with the file specified. It can be one of the following:</p> <p>SHARED Any process on any machine may read or write the file.</p> <p>LOCK READ No other process is to be granted read access to this file. This access will be granted only if no other process has LOCK READ access to the file.</p> <p>LOCK WRITE No other process is to be granted write access to this file. This access will be granted only if no other process has LOCK WRITE access to the file.</p> <p>LOCK READ WRITE No other process is to be granted read or write access to this file. This access will be granted only if no other process has LOCK READ or WRITE access to the file.</p> <p>If access is not specified, the file may be opened for reading and writing any number of times by the</p>

COMMANDS, STATEMENTS AND FUNCTIONS

SYNTAX ELEMENT	MEANING
	<p>process, but other processes are denied access to this file while it is open. See also the LOCK and UNLOCK statements.</p> <p>Note that if you try and use this parameter with an MS-DOS release prior to 3.1 you will get an "Advanced Feature Error". If you are using MS-DOS 3.1 or later, but MS-NET is not installed, you will get a "Permission Denied". The MS-DOS command SHARE must also be given (either at the MS-DOS prompt or from an AUTOEXEC.BAT file). See the "MS-DOS User Guide" for a full description of procedures to be followed for networking.</p>
<i>filenum</i>	<p>Is an integer expression returning a number in the range 1 through 255. The number is used to associate an I/O buffer with a disk file or device. This association exists until a CLOSE or CLOSE <i>filenum</i> statement is executed. The file is referred to by this number in any I/O statement by this number.</p>
<i>record-length</i>	<p>Is an integer expression from 1 to 32767. This value sets the record length to be used for random files. If omitted, the <i>record-length</i> defaults to 128 byte records. The <i>record-length</i> may not be greater the value specified by the /S: switch in the GWBASIC command. GW-BASIC will ignore this option if it is used to OPEN a sequential file.</p>
<i>mode2</i>	<p>Is a string expression whose first character is one of the following:</p> <ul style="list-style-type: none"> O Specifies sequential output mode I Specifies sequential input mode R Specifies random input/output mode A Specifies sequential output mode and sets the file pointer at the end of the file, and the record number as the last record of the file. A PRINT # or WRITE # statement will then extend (append) the file.

Characteristics

A disk file must be opened before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file or device and determines the mode of access that will be used with the buffer. The *filenum* parameter specifies the number which will be associated with the file as long as it is open and will be used by other I/O statements to refer to the file or device.

The maximum number of files that may be open simultaneously is set by the /F: switch in the GWBASIC command. This number falls within the range 1 to 15, and defaults to 3.

For each device or file, the following OPEN modes are allowed:

KYBD:	INPUT only
SCRN:	OUTPUT only
LPT1:	OUTPUT or random (*)
LPT2:	OUTPUT or random (*)
LPT3:	OUTPUT or random (*)
COM1:, COM2:, Disk files	INPUT or OUTPUT INPUT, OUTPUT, APPEND or random

(*) GW-BASIC will not send a line feed after each carriage return, if a printer has been opened in random mode with a width of 255.

The GW-BASIC file I/O system allows the user to take advantage of user installed devices.

Character devices are opened and used in the same manner as disk files. However, characters are not buffered by GW-BASIC as they are for disk files. The record length is set to one.

GW-BASIC only sends a CR (carriage return X'0D') as end of line. If the device requires a LF (line feed X'0A'), the driver must provide it. When writing device drivers, keep in mind that GW-BASIC users will want to read and write control information. Writing and reading of device control data is handled by the GW-BASIC IOCTL statement and IOCTL\$(f) function.

COMMANDS, STATEMENTS AND FUNCTIONS

Remarks

1. If you enter a value outside of the corresponding range an "Illegal function call" error is returned, and the file will not be opened.
2. If the file is opened for INPUT, attempts to write to the file will result in a "Bad file mode" error. If a file opened for input does not exist, a "File not found" error occurs.
3. When a disk file is opened for APPEND, the pointer position is initially at the end of the file and the record is set to the last record of the file. PRINT #, or WRITE # will then extend the file.
4. If the file is opened for OUTPUT or APPEND, attempts to read the file will result in a "Bad file mode" error.
5. If you open a file which does not exist for output, append, or random access, you will create that file.
6. A file can be opened for sequential input or random access on more than one file number at a time. A file may NOT be opened for OUTPUT or APPEND, on more than one file number at a time.

Moreover, since it is possible to reference the same file in a sub-directory via different pathnames, it is impossible for GW-BASIC to know that it is the same file simply by looking at the pathname. For this reason, GW-BASIC will not let you open the file for OUTPUT or APPEND if it is on the same disk, even if the pathname is different.

Examples

```
10 OPEN "I",2,"INVEN"
```

```
10 OPEN "MAILING.DAT" FOR APPEND AS 1
```

```
10 OPEN "CON:" FOR INPUT LOCK READ AS #1
```

```
10 OPEN "MYFILE" FOR OUTPUT LOCK WRITE AS #2
```


If you write and install a device called F01, then the OPEN statement might appear as:

```
10 OPEN "\DEV\F01" FOR OUTPUT AS #1
```

To open the printer for output, the user could use the line:

```
100 OPEN "LPT:" FOR OUTPUT AS #1
```

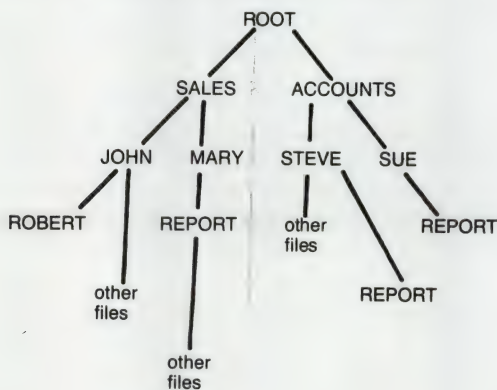
which uses the GW-BASIC device driver, or as part of a pathname as in:

```
100 OPEN "\DEV\LPT1" FOR OUTPUT AS #1
```

which uses the MS-DOS device driver.

Examples

Using the following tree structure:



If MARY is your current directory, then:

```
OPEN "REPORT" ...
OPEN "\SALES\MARY\REPORT" ...
OPEN "..\MARY\REPORT" ...
```

all refer to the same file.

COMMANDS, STATEMENTS AND FUNCTIONS

Possible Errors

"Bad file name"

"Bad file number"

"Bad file mode"

"Too many files": too many files are open. (See the /F: switch in the GWBASIC command line).

"File not found": if a file opened for input does not exist.

"Device not available": you have attempted to open either a directory, or a non-existent device.

"File already open"

"Device I/O error": reception error. Usually caused by an incorrectly written device driver (user-installed).

"Illegal function call": usually caused by an excessive record length. (See the /S: switch in the GWBASIC command line).

"Permission Denied": failed attempt to open a file, due to restriction of access.

"Path/File Access Error": failed attempt to open a file, due to mode specified being incompatible with network-installed sharing access to a device.

OPEN COM Statement

Opens and initializes an RS-232-C port for input/output.

```
OPEN "COM n : [speed] [ , [parity]] [ , [data] [ , [stop] [,RS][,CS [t]]  
[ ,DS [t]][ ,CD [t]][ ,BIN ][ ,ASC ][ ,LF ]]" [ FOR mode ] AS [ # ] filenum  
[ LEN = record-length]
```

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is 1 or 2. It specifies the number of the RS-232-C port.
<i>speed</i>	Is an integer constant which sets the baud rate in bits per second of the specified port. Valid values are: 75, 110, 150, 300, 600, 1200, 1800, 2400, 4800, and 9600. The default value is 300 bps.
<i>parity</i>	Designates the parity. Valid entries are: E (even) - default value M (mark) N (none)

COMMANDS, STATEMENTS AND FUNCTIONS

SYNTAX ELEMENT	MEANING
	O (odd) S (space)
<i>data</i>	Designates the number of data bits per byte. Valid entries are: 5, 6, 7 (default), or 8.
<i>stop</i>	Designates the stop bit. Valid entries are: 1, 1.5, or 2. If omitted then 75 and 110 bps transmit two stop bits, all others transmit one stop bit.
RS	Suppresses RTS (Request To Send)
CS[t]	Controls CTS (Clear To Send)
DS[t]	Controls DSR (Data Set Ready)
CD[t]	Controls CD (Carrier Detect)
BIN	Opens the file in binary mode. BIN is selected by default, unless ASC is specified.
ASC	Opens the file in ASCII mode.
LF	Specifies that a linefeed is to be sent after a carriage return.

SYNTAX ELEMENT	MEANING
<i>mode</i>	<p>Is one of the following string expressions:</p> <p>OUTPUT Specifies sequential output mode</p> <p>INPUT Specifies sequential input mode</p> <p>If the <i>mode</i> expression is omitted, it is assumed to be random input/output. Random cannot, however, be explicitly chosen as <i>mode</i>.</p>
<i>filenum</i>	Is the number of the file to be opened.
<i>record-length</i>	<p>Is the length of the records written to or read from a communications buffer. This value cannot be greater than the value fixed by the /C: switch in the GWBASIC command. The default <i>record-length</i> for the receive buffer is 2 bytes. The length of the transmit buffer is 128 bytes.</p>

Characteristics

The OPEN COM statement must be executed before an RS-232-C port can be used.

Any syntax errors in the OPEN COM statement will result in a "Bad file name" error message.

The *speed*, *parity*, *data*, and *stop* options must be listed in the order shown in the above syntax. The remaining options may be listed in any order, but they must be listed after the *speed*, *parity*, *data*, and *stop* options.

COMMANDS, STATEMENTS AND FUNCTIONS

The CS, DS, and CD options allow you to specify a time (*t*) to wait for the signal before a "Device Timeout" error is returned. This time is expressed in milliseconds, ranging from 0 to 65535. Default values are: CD = 0, CS = 1000, DS = 1000.

LF allows communication files to be printed on a serial line printer. When LF is specified, a linefeed character (0AH) is automatically sent after each carriage return character (0DH). This includes the carriage return sent as a result of the width setting. Note that INPUT # and LINE INPUT #, when used to read from a COM file that was opened with the LF option, stop when they see a carriage return, ignoring the linefeed.

The LF option is superseded by the BIN option.

In the BIN mode, tabs are not expanded to spaces, a carriage return is not forced at the end-of-line, and CTRL Z is not treated as end-of-file. When the file is closed, CTRL Z will not be sent over the RS-232 line. The BIN option supersedes the LF option.

In ASC mode, tabs are expanded, carriage returns are forced at the end-of-line, CTRL Z is treated as end-of-file, and XON/XOFF protocol is enabled. When the channel is closed, CTRL Z will be sent over the RS-232 line.

Example

```
10 OPEN "COM1:9600,N,8,1,BIN" AS #2
```

will open communications port 1 in random mode at a speed of 9600 baud with no parity bit, 8 data bits, and 1 stop bit. Input/Output will be in binary mode. Other lines in the program may now access port 1 as file number 2.

OPTION BASE Statement

Defines the minimum value for array subscripts.

OPTION BASE n

Where

SYNTAX ELEMENT	MEANING
n	Is an integer expression and may be 1 or 0

Characteristics

The default base is 0. If the statement:

OPTION BASE 1

is executed, the lowest value an array subscript may have is 1.

A CHAINED program may have an OPTION BASE statement if no arrays are passed. The CHAINED program will inherit the OPTION BASE value of the chaining program.

COMMANDS, STATEMENTS AND FUNCTIONS

Possible Errors

The OPTION BASE statement must be executed before definition or usage of arrays. A "Duplicate Definition" error occurs when the base value is changed when arrays are in existence.

OUT Statement

Transmits a byte to an output port.

OUT *port* , *byte*

Where

SYNTAX ELEMENT	MEANING
<i>port</i>	Is an integer expression in the range 0 through 65535 and represents an output port number.
<i>byte</i>	Is an integer expression in the range 0 through 255 and represents the data to be transmitted.

Characteristics

OUT is the complimentary statement to the INP statement.

If *port* or *byte* is outside the specified range, an "Illegal function call" error is returned.

PAINT must start on a non-border point, otherwise it will have no effect. Nor will it have any effect if the specified point already has the *border* color.

PAINT can fill any figure, but **PAINTing** complex figures may result in an "Out of Memory" error. If this happens, use the **CLEAR** statement to increase the amount of stack space available, and re-execute the program.

Tiling

Using the **PAINT** statement you can paint an area using various colors for the pixels, in order to achieve a "structure of colors" (or tiling). In order to use this feature, you must provide a string value for the *paint* parameter, according to the following syntax:

PAINT(*x* , *y*), **CHR\$** (*n*) [+ **CHR\$**(*n*)]...

where *n* is a number between 0 and 255, or between &H00 and &HFF in hexadecimal. You can specify up to 64 **CHR\$** (*n*) functions in each **PAINT** statement. The sequence of **CHR\$** (*n*) string functions allows you to create a mask (or "tiling patch") with a rectangular structure of 8 bits in width and a variable height (up to 64 bits). This mask defines the basic tiling structure.

The mask (which is repeated uniformly over the area to be painted and clipped at the borders of the area) is as follows:

	$x \rightarrow$	
$y \downarrow$	x x x x x x x x	CHR\$ (<i>n</i>) byte 0
	x x x x x x x x	CHR\$ (<i>n</i>) byte 1
	x x x x x x x x	CHR\$ (<i>n</i>) byte 2
	.	.
	.	.
	.	.
	x x x x x x x x	CHR\$ (<i>n</i>) byte 63 (maximum permitted)

COMMANDS, STATEMENTS AND FUNCTIONS

The horizontal repetition of the mask on the screen is carried out in such a way that each line of the mask corresponds with one byte of the screen memory, that is 4 pixels in Medium Resolution and 8 pixels in High and Super Resolution. Since there are 80 bytes associated with the entire screen, there are 80 possible horizontal positions that the mask can take. Vertically, the mask starts at the top margin of the screen and is repeated towards the bottom margin of the screen. This means that a mask of height h is repeated vertically every h scanlines, where the first line of the mask is always placed on a scanline (y-coordinate) that is a multiple of h .

In order to calculate which line of the mask corresponds with a given scanline you must use the following formula:

$$\text{line} = \text{scanline} \text{ MOD } h$$

Where

line is the line of the mask; 0 is the topmost line

scanline is the scanline (y-coordinate); 0 is the first scanline

h is the height of the mask

Using the following statement:

```
PAINT (320, 100) , CHR$ (&H81) + CHR$ (&H42) + CHR$
(&H24) + CHR$ (&H18) + CHR$ (&H18) + CHR$
(&H24) + CHR$ (&H42) + CHR$ (&H81)
```

tiling begins at the point (320, 100), and the first scanline of the mask used is line 4, since $\text{line} = 100 \text{ MOD } 8$.

Using the above statement you can tile the screen with x's, in High and Super Resolution; the mask is constructed as follows:

	x→	
y	1 0 0 0 0 0 1	CHR\$ (&H81) byte 0
↓	0 1 0 0 0 0 1 0	CHR\$ (&H42) byte 1
	0 0 1 0 0 1 0 0	CHR\$ (&H24) byte 2
	0 0 0 1 1 0 0 0	CHR\$ (&H18) byte 3
	0 0 0 1 1 0 0 0	CHR\$ (&H18) byte 4
	0 0 1 0 0 1 0 0	CHR\$ (&H24) byte 5
	0 1 0 0 0 0 1 0	CHR\$ (&H42) byte 6
	1 0 0 0 0 0 0 1	CHR\$ (&H81) byte 7

The same PAINT statement, in Medium Resolution, paints the screen with a red and green pixel pattern (if the active palette is palette 0).

In some cases you may want to tile a figure which is already painted; if the area of the figure being tiled has the same color as two consecutive characters in the mask, tiling will be terminated. It is possible to overcome this limitation by using the *background* parameter; in order to specify that a color should not be considered as a stop condition, you have to assign to *background* the value of the two consecutive mask characters. If palette 0 was active for example, and you had assigned CHR\$ (&HAA) to *background* (that is red), it would be possible to draw two red lines and two blue lines, alternatively, onto a background color of red.

If you specify more than two consecutive bytes in the mask that match *background* you will get the error message "Illegal function call"; this means that it is not possible to draw, for example, more than two red lines on an area already painted red.

Example

```
10 SCREEN 1
20 COLOR 0,0,1,0
30 CLS
40 CIRCLE (256,128),130,2
50 PAINT (256,128),1,2
60 LINE (251,123)-STEP(10,10),0,BF
```

Statement 10 selects Medium Resolution. Statement 20 selects black for color number 0, palette 0 (green, red, yellow), green as graphics foreground, black as graphics background. Statement 30 clears the screen with the background color (in this case black). Statement 40 draws a red circumference with a radius of 130 whose center is (256,128). Statement 50 paints the circle green. Statement 60 draws a black filled-in box in the middle of the circle.

PEEK Function

Returns the byte read from the specified memory location.

PEEK(*offset*)

Where

SYNTAX ELEMENT	MEANING
<i>offset</i>	Is a numeric expression returning an integer in the range 0 to 65535. It indicates the address of the memory location of the byte that PEEK will read. It is the offset from the current segment, which was defined by the last DEF SEG statement (if no DEF SEG statement has been executed the current segment is the GW-BASIC Data Segment).

Characteristics

The returned value is an integer in the range 0 to 255.

If *offset* is outside the specified range, an "Illegal function call" error is returned.

PEEK is the complementary function of the POKE statement.

Example

A = PEEK(&H5A00)

PLAY Statement

Plays music in accordance with a string which specifies the notes to be played, and the way in which the notes are to be played.

PLAY *stringexp*

Where

SYNTAX ELEMENT	MEANING
<i>stringexp</i>	Is a string expression containing a series of single-character commands

Characteristics

PLAY uses a concept similar to that in DRAW by embedding a Music Macro Language into one statement. A set of subcommands, used as part of the PLAY statement, specifies the particular action to be taken.

COMMANDS, STATEMENTS AND FUNCTIONS

The subcommands used for *stringexp* are:

COMMAND	ACTION
A-G[# + -]	Plays a note in the range A-G. The suffixes (#) or (+) after the note specify sharp; suffix (-) specifies flat.
On	Sets the current octave. There are seven octaves, numbered 0 through 6.
>n	Increments the octave and plays note <i>n</i> . The octave is progressively incremented, each time note <i>n</i> is played, until octave 6 is reached. Note <i>n</i> is subsequently played at octave 6.
<n	Decrements the octave and plays note <i>n</i> . The octave is progressively decremented, each time note <i>n</i> is played, until octave 0 is reached. Note <i>n</i> is subsequently played at octave 0.
Nn	Plays one of 84 notes within the 7 possible octaves. The <i>n</i> parameter ranges from 0 to 84; 0 indicates a rest. This command is an alternative to specifying notes using the note name (A-G) and octave number commands.
Pn	Specifies a pause. The <i>n</i> parameter ranges from 1 to 64 and corresponds to the length of each note, set by <i>Ln</i> .

COMMAND	ACTION
Ln	<p>Sets the length of each note. The n parameter ranges from 1 to 64, where $n = 1$ is equivalent to a whole note; $n = 4$ is equivalent to a quarter note, etc.</p> <p>The length may also follow the note when a change of length only is required for a particular note. In this case, A16 is equivalent to L16A.</p>
.	<p>A period after a note causes the note to be played $3/2$ times the length determined by L multiplied by T (tempo). Multiple periods may appear after a note. The period is scaled accordingly; e.g., A. is $3/2$, A.. is $9/4$, A... is $27/8$, etc. Periods may appear after a pause (P). In this case, the pause length may be scaled in the same way notes are scaled.</p>
Tn	<p>Sets the tempo, or number of quarter notes in one minute. The n parameter ranges from 32 to 255, with a default value of 120.</p>
MF	<p>Sets Music Foreground. The PLAY and SOUND statement are to run in foreground. Each successive note or sound will not start until the preceding note or sound has finished. This is the default setting.</p>
MB	<p>Sets Music Background. The PLAY and SOUND statement are to run in background. That is, each note or sound is placed in a buffer allowing the GW-BASIC program to continue executing while the note or sound plays in the "background". Up to 32 notes can be played in the background at a time.</p>

COMMANDS, STATEMENTS AND FUNCTIONS

COMMAND	ACTION
MN	Sets "music normal", so that each note will play $\frac{7}{8}$ of the time determined by length (L).
ML	Sets "music legato", so that each note will play the full period set by length (L).
MS	Sets "music staccato", so that each note will play $\frac{3}{4}$ of the time set by length (L).
X <i>string</i>	Executes the specified string.

Remarks

The *n* parameter may be constant or variable, where a variable is written as *= variable*;. The semicolon is necessary when a variable is used in this way, or when the X command is used, but it is not allowed after MF, MB, MN, ML, or MS. In all other cases, a semicolon is optional between commands.

Example

```
10 LET LISTEN$ = "T180 O2 P2 P8 L8 GGG L2 E-"  
20 LET FATE$ = "P24 P8 L8 FFF L2 D"  
30 PLAY LISTEN$ + FATE$
```

This example will play the beginning of the first movement of Beethoven's Fifth Symphony.

Examples

100 PLAY "<<" 'Decrement by two octaves

200 PLAY ">" 'Increment by an octave

300 PLAY "A>" 'Increment by an octave and play an A note

400 PLAY "XSONG\$"

PLAY(*n*) Function

Returns the number of notes remaining in the music background buffer.

PLAY(*n*)

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is a dummy argument. Any value may be supplied.

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

IF...	THEN
The program is running in Music Foreground mode	PLAY(<i>n</i>) returns 0
The program is running in Music Background mode	PLAY(<i>n</i>) returns the number of notes currently in the Music Background buffer. The maximum value that PLAY(<i>n</i>) may return is 32.

Example

10 IF PLAY(0) = 6 GOTO 500

PLAY { ON | OFF | STOP } Statements

PLAY ON enables PLAY(*n*) trapping.
PLAY OFF disables PLAY(*n*) trapping.
PLAY STOP suspends PLAY(*n*) trapping.

PLAY { ON | OFF | STOP }

Characteristics

PLAY ON, PLAY OFF, PLAY STOP are used in conjunction with the ON PLAY(*n*) GOSUB statement.

If a PLAY OFF statement has been executed the GOSUB is not performed and is not remembered.

If a PLAY STOP statement has been executed the GOSUB is not performed, but will be performed as soon as PLAY ON statement is executed.

If an ON PLAY(*n*) GOSUB is performed, an automatic PLAY STOP is executed.

Once an error trap takes place, all trapping is automatically disabled.

The RETURN *linenum* form of the RETURN statement may be used to return to a specific line number from the trapping subroutine. Use this type of return with care, however, because any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active, and errors such as "FOR without NEXT" may result.



PMAP Function

Converts screen coordinates to world coordinates or vice versa. (Graphics Mode only).

PMAP(*coordinate* , *n*)

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
<i>coordinate</i>	Is a numeric expression specifying either the <i>x</i> coordinate or the <i>y</i> coordinate of the point to be mapped according to the value of <i>n</i> .
<i>n</i>	<p>Is an integer number in the range 0 to 3:</p> <p>0 specifies that the <i>coordinate</i> value is the world <i>x</i> coordinate, which is to be mapped to the screen <i>x</i> coordinate</p> <p>1 specifies that the <i>coordinate</i> value is the world <i>y</i> coordinate, which is to be mapped to the screen <i>y</i> coordinate</p> <p>2 specifies that the <i>coordinate</i> value is the screen <i>x</i> coordinate, which is to be mapped to the world <i>x</i> coordinate</p> <p>3 specifies that the <i>coordinate</i> value is the screen <i>y</i> coordinate, which is to be mapped to the world <i>y</i> coordinate</p>

Characteristics

The four PMAP functions allow you to find equivalent point locations between those in the window created with the WINDOW statement and those in the viewport as defined by the VIEW statement.

Examples

If a user had defined a WINDOW SCREEN (80,100) - (200,200) then the upper left coordinate of the window would be (80,100) and the lower right would be (200,200). With screen2 the coordinates are (0,0) in the upper left hand corner and (639,199) in the lower right. Then:

X = PMAP(80,0)

would return the screen x coordinate that corresponds to the world x coordinate 80:

0

The PMAP function in the statement:

Y = PMAP(200,1)

would return the screen y coordinate that corresponds to the world y coordinate 200:

199

The PMAP function in the statement:

X = PMAP(639,2)

would return the world x coordinate that corresponds to the screen x coordinate 639:

200

The PMAP function in the statement:

Y = PMAP(199,3)

would return the world y coordinate that corresponds to the screen y coordinate 199:

200

POINT Function

Returns the color number of a pixel on the screen, if two arguments (x,y) are given, or the x- or y-coordinate of the last referenced point if one argument (n) is given (Graphics Mode only).

Syntax 1

POINT(x , y)

Syntax 2

POINT(n)

Where

SYNTAX ELEMENT	MEANING
x,y	Are the absolute screen coordinates of the selected pixel. If the point is outside the current viewport, the value -1 is returned.

SYNTAX ELEMENT	MEANING
<i>n</i>	<p>May have the following values:</p> <ul style="list-style-type: none"> 0 Specifies the current screen x coordinate 1 Specifies the current screen y coordinate 2 Specifies the current world x coordinate if a WINDOW statement has been executed, otherwise specifies the same value as the POINT(0). 3 Specifies current world y coordinate if a WINDOW statement has been executed, otherwise specifies the same value as the POINT(1).

Characteristics

$v\% = \text{POINT}(x,y)$

returns the color number of the specified pixel into the integer variable $v\%$.

In Medium Resolution, POINT (*x*, *y*) can return 0, 1, 2 or 3; in High and Super Resolution, it can return 0 or 1.

$v2 = \text{POINT}(1)$

returns the screen coordinate of the current point into the single (or double) precision variable $v2$.

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
10 SCREEN 2
20 IF POINT(I,I) < > 0 THEN PRESET (I,I) ELSE PSET (I,I)
30 'Invert point for B/W system
40 PSET (I,I), 1-POINT(I,I)
```

Example

```
10 SCREEN 1
20 LET C = 3
30 PSET (10,10), C
40 IF POINT(10,10) = C THEN PRINT "This point is color "; C
```


POKE Statement

Writes a byte into a memory location.

POKE *offset* , *byte*

Where

SYNTAX ELEMENT	MEANING
<i>offset</i>	Is a numeric expression returning an integer in the range 0 to 65535 and indicates the address of the memory location where the byte is to be written by the POKE statement. It is the offset from the current segment, which was set by the last DEF SEG statement. If no DEF SEG statement has been executed, the current segment is the GW-BASIC Data Segment.
<i>byte</i>	Is the byte to be written to the specified address . It must be in the range 0 to 255.

Characteristics

You can use POKE and PEEK for passing arguments and data to machine language subroutines. If either *offset* or *byte* is outside the specified range, an "Illegal function call" error is returned. The complementary function to POKE is PEEK.

COMMANDS, STATEMENTS AND FUNCTIONS

Warning

Use POKE carefully. If it is used incorrectly, it can cause GW-BASIC or MS-DOS to crash.

Example

```
10 POKE &H5A00,&HFF
```

POS Function

Returns the current horizontal (column) position of the cursor.

POS(*dummy*)

Where

SYNTAX ELEMENT	MEANING
<i>dummy</i>	Is a dummy argument. Any value is accepted.

Characteristics

The current horizontal (column number) position of the cursor is returned. The leftmost position is 1. The rightmost position may be 40, or 80, depending on the current screen width. To return the current vertical line position of the cursor (row number), use the CSRLIN function.

Example

```
IF POS(0) > 70 THEN BEEP
```

PRESET Statement

Draws or erases a point at the specified position on the screen (Graphics Mode only).

```
PRESET [ STEP ] ( x , y ) [ , color ]
```

Where

SYNTAX ELEMENT	MEANING
<i>x,y</i>	Are the coordinates of the point. They may be in absolute or relative form (if the STEP option is included)
<i>color</i>	Is an integer expression representing the color number of the specified point, in the range 0 to 3. In Medium Resolution <i>color</i> is chosen from the active palette; in High and Super Resolution, the values 0 and 2 indicate black and 1 and 3 indicate white. If no <i>color</i> parameter is given, the graphics background color is selected. If <i>color</i> is included, PRESET has the same meaning as PSET.

Example

PRESET (x,y)

Erases the point at the given coordinates from the screen; it has the same meaning as:

PSET (x,y),0

if the graphics background color is 0.

Possible Errors

If a color greater than 3 is given this will result in an "Illegal function call".

PRINT Statement

Outputs data to the screen.

PRINT [*list-of-expressions*[, | ;]]

Where

SYNTAX ELEMENT	MEANING
<i>list-of-expressions</i>	May be numeric and/or string expressions. (String constants must be enclosed in quotation marks.) Each expression should be separated from the next by a comma, blank, or semicolon.

Characteristics

If you omit the *list-of-expressions*, a blank line is displayed. If you include the *list-of-expressions*, the values of the expressions are displayed on the screen.

The position of each printed item is determined by the punctuation used to separate the items in the list. GW-BASIC divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT or PRINT USING statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, GW-BASIC goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 7 or fewer digits in the unscaled format, are output using the unscaled format. For example, 1E-7 is output as .0000001 and 1E-8 is output as 1E-08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D-16 is output as .00000000000000001 and 1D-17 is output as 1D-17.

A question mark may be used in place of the word PRINT in a PRINT statement.

COMMANDS, STATEMENTS AND FUNCTIONS

Example 1

```
10 X=5
20 PRINT X+5,X-5,X*(-5),X^5
30 END
RUN
10      0      -25      3125
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2

```
LIST
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
  9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
  21 SQUARED IS 441 AND 21 CUBED IS 9261

?
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line. Line 40 causes a blank line to be printed before the next prompt.

Example 3

```
10 FOR X=1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
```

FORMATTING CHARACTER	MEANING
&	<p>Specifies a variable length string field. When the field is specified with "&", the string is output without modification.</p> <p>For example:</p> <pre>10 A\$ = "LOOK":B\$ = "OUT" 20 PRINT USING "!";A\$; 30 PRINT USING "&";B\$ RUN LOUT</pre>

Numeric Fields

When PRINT USING is used to print numbers, the formatting special characters may be used to format the numeric field:

FORMATTING CHARACTERS	MEANING
#	<p>Represents each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.</p>

COMMANDS, STATEMENTS AND FUNCTIONS

FORMATTING CHARACTER	MEANING
	<p>A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0, if necessary). Numbers are rounded as necessary.</p> <p>PRINT USING " # . # # ";.78 0.78</p> <p>PRINT USING " # # . # # ";987.654 987.65</p> <p>PRINT USING " # # . # # ";10.2,5.3,66.789,.234 10.20 5.30 66.79 0.23</p> <p>In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.</p>
+	<p>A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.</p>
-	<p>A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.</p> <p>PRINT USING " + # # . # # ";-68.95,2.4,55.6,-.9 -68.95 +2.40 +55.60 -0.90</p> <p>PRINT USING " # # . # # - ";-68.95,22.449,-7.01 68.95- 22.45 7.01-</p>

FORMATTING CHARACTER	MEANING
**	<p>A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.</p> <p>PRINT USING "***#. # ";12.39,-0.9,765.1 *12.4 *-0.9 765.1</p>
\$\$	<p>A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The \$\$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with \$\$. Negative numbers cannot be used unless the minus sign trails to the right.</p> <p>PRINT USING "\$\$###. # #";456.78 \$456.78</p>
\$	<p>The **\$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. **\$ specifies three more digit positions, one of which is the dollar sign.</p> <p>PRINT USING "*\$###. # #";2.34 ***\$2.34</p>

COMMANDS, STATEMENTS AND FUNCTIONS

FORMATTING CHARACTER	MEANING
	<p>A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^ ^ ^ ^) format.</p> <p>PRINT USING "###,.#";1234.5 1,234.50</p> <p>PRINT USING "###.##";1234.5 1234.50,</p>
^ ^ ^ ^	<p>Four carets (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carets allow space for E + xx or D + xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.</p> <p>PRINT USING "###.## ^ ^ ^ ^";234.56 2.35E + 02</p> <p>PRINT USING ".### ^ ^ ^ ^-";888888 .8889E + 06</p> <p>PRINT USING "+.## ^ ^ ^ ^";123 +.12E + 03</p>

FORMATTING CHARACTER	MEANING
_	<p>An underscore in the format string causes the next character to be output as a literal character.</p> <p>PRINT USING "_!#.#.#_!";12.34 !12.34!</p> <p>The literal character itself may be an underscore by placing "__" in the format string.</p>
%	<p>If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.</p> <p>PRINT USING "#.#.#";111.22 %111.22</p> <p>PRINT USING ".#.#";.999 %1.00</p> <p>If the number of digits specified exceeds 24, an "Illegal function call" error will result.</p>

COMMANDS, STATEMENTS AND FUNCTIONS

PRINT # and PRINT # USING Statements

Write data sequentially to a disk file. The PRINT # and PRINT # USING statements are usually used in a program.

PRINT # *filenum* , [**USING** *format-string* ;]*list-of-expressions*

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is the number used when the file was OPENed with the option OUTPUT specified
<i>format-string</i>	Is a string expression (usually a constant or variable) composed of formatting characters described in the "PRINT USING" statement
<i>list-of-expressions</i>	Is a list of the numeric and/or string expressions to be written to file

Characteristics

PRINT # does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal screen with a PRINT statement. For this reason, care should be taken to delimit the data on the disk, so that it will be input correctly from the disk.

In the list of expressions, numeric expressions should be delimited by semicolons. For example:

```
50 PRINT # 1,B;C;D;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to the disk.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, let A\$ = "CAMERA" and B\$ = "93604-1". The statement

```
100 PRINT # 1,A$;B$
```

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT # statement as follows:

```
200 PRINT # 1,A$;" ";B$
```

The image written to disk is:

```
CAMERA,93604-1
```

which can be read back into two string variables. If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to disk surrounded by explicit quotation marks, using CHR\$(34). For example, let A\$ = "CAMERA, AUTOMATIC" and B\$ = " 93604-1". The statement:

```
300 PRINT # 1,A$;B$
```

would write the following image to disk:

```
CAMERA, AUTOMATIC 93604-1
```

COMMANDS, STATEMENTS AND FUNCTIONS

And the statement:

```
400 INPUT #1,A$,B$
```

would input "CAMERA" to A\$ and "AUTOMATIC 93604-1" to B\$. To separate these strings properly on the disk, write double quotation marks to the disk image using CHR\$(34). The statement:

```
500 PRINT #1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to disk:

```
"CAMERA, AUTOMATIC"" 93604-1"
```

And the statement:

```
600 INPUT #1,A$,B$
```

would input "CAMERA, AUTOMATIC" to A\$ and " 93604-1" to B\$. The PRINT # statement may also be used with the USING option to control the format of the disk file. For example:

```
700 PRINT #1,USING"$$# # #.# #,";J;K;L
```

For more examples using PRINT #, see Chapter 4 (Disk File Handling). See also "WRITE#", later in this chapter.

PSET Statement

Illuminates a pixel at a specified position with a given color, on the screen. (Graphics Mode only).

PSET [STEP] (x , y) [, color]

Where

SYNTAX ELEMENT	MEANING
<i>x,y</i>	Are the coordinates of the pixel to be illuminated. You may specify them either in absolute or relative form.
<i>color</i>	Is an integer expression whose value is in the range 0 to 3. It represents the color number of the point specified. In Medium Resolution <i>color</i> is chosen from the active palette; in High and Super Resolution the values 0 and 2 represent black and 1 and 3 represent white. If <i>color</i> is not specified the graphics foreground color is selected.

Characteristics

Coordinates can be specified in one of two forms:

PSET STEP (*x-offset*, *y-offset*) or
PSET (*absolute-x*, *absolute-y*)

The first form is a point relative to the most recent point referenced. The second form is more common and directly refers to a point without regard to the last point referenced.

Examples are:

PSET (10,10)	absolute form
PSET STEP (10,0)	offset 10 in x and 0 in y
PSET (0,0)	origin

COMMANDS, STATEMENTS AND FUNCTIONS

Examples

This example draws a diagonal line from (0,0) to (100,100):

```
10 FOR i=0 TO 100
20 PSET (i,i)
30 NEXT
```

This example erases the line by setting each pixel to 0:

```
40 FOR i=100 TO 0 STEP -1
50 PSET (i,i),0
60 NEXT
```

PUT (COM files) Statement

Writes a specified number of bytes to a communications file. PUT (COM files) is usually used in a program.

PUT [#] *filenum* [, *length*]

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is an integer expression returning a valid file number

SYNTAX ELEMENT	MEANING
<i>length</i>	Is an integer expression returning the number of bytes to be transferred out of the communications buffer. <i>length</i> cannot exceed the value specified by the LEN clause in the OPEN COM statement.

Example

100 PUT #2,80

PUT (Files) Statement

Writes a record from a random buffer to a random file. PUT (Files) is usually used in a program.

PUT [#]*filenum*[, *recordnum*]

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is the number under which the file was OPENed

COMMANDS, STATEMENTS AND FUNCTIONS

SYNTAX ELEMENT	MEANING
<i>recordnum</i>	The number of the record in the file. It must be in the range 1 to 16,777,215. If omitted the current record number is assumed (i.e., the record whose number is one higher than that of the last record accessed).

Characteristics

PRINT #, PRINT# USING, WRITE #, LSET and RSET may be used to put characters in the random file buffer before executing a PUT statement.

In the case of WRITE #, GW-BASIC pads the buffer with spaces up to the carriage return.

Example

LIST

```
10 OPEN "r",1,"A:RAND",48
20 FIELD 1,20 AS R1$,20 AS R2$,8 AS R3$
30 FOR L=1 TO 4
40 INPUT "name";N$
50 INPUT "address";M$
60 INPUT "phone";P$
70 LSET R1$ = N$
80 LSET R2$ = M$
90 LSET R3$ = P$
100 PUT 1,L
110 NEXT L
120 CLOSE 1
130 END
Ok
```

RUN

name? **super man**

address? **USA**

phone? **11234621**

name? **robin hood**

address? **England**

phone? **23462101**

.

.

.

Ok

Statement 10 opens the random file RAND, with a record length of 48 on the diskette inserted in drive A. The file number is 1. Statement 20 divides the buffer into fields.

Statement 100 writes a record to file RAND, with the record number being set by the control variable of the FOR/NEXT loop.

Possible Errors

Any attempt to read or write past the end of the buffer causes a "Field overflow" error.



PUT (Graphics) Statement

Transfers the graphics image stored in an array by a GET statement, to the screen.

PUT (x, y), array[, action-verb]

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
<i>x,y</i>	Represent the top left corner of the rectangle to be displayed
<i>array</i>	Is the name of an array containing the image to be displayed. The type of the array must be numeric.
<i>action-verb</i>	Is a string of characters that specify a logical operation between the colors of the points to be displayed and the colors of the points already on the screen. The string can be one of the following: PSET, PRESET, AND, OR, XOR. The default <i>action-verb</i> is XOR.

Remarks

The PUT statement is used to transfer an image onto the screen, taking it from an array in memory (the image must have already been stored using a GET statement): it also allows the animation of objects.

The array, which can be of any numeric type, holds the image, and must have been given dimensions large enough to hold the colors of all the pixels of the image.

The Action Verb Parameter

The *action-verb* specifies the logical operation to be carried out between the colors of points to be displayed and the colors of the pixels already on the screen.

It can have one of the following values:

- PSET, transfers the data pixel by pixel onto the screen. Each pixel has the exact color it had when it was memorized from the screen.
- PRESET, is the same as PSET except that a negative image is produced.
- AND is used when the image is to be transferred over an existing image on the screen. Points that had the same color in both the existing image and the image to be displayed will remain the same color, points that do not have the same color are transformed using the AND operation between corresponding bits.
- OR, is used to superimpose the image onto an existing image.
- XOR, is used for animation. It causes a point on the screen to be inverted if there is a corresponding point in the image stored in the array. When an image is displayed on a complex background twice in the same position, the image is erased without altering the background color. This allows you to move an object around the screen without altering the background colors.

In Medium Resolution AND, OR and XOR have the following effects on color:

AND

screen

		0	1	2	3
array	0	0	0	0	0
	1	0	1	0	1
value	2	0	0	2	2
	3	0	1	2	3

OR

screen

		0	1	2	3
array	0	0	1	2	3
	1	1	1	3	3
value	2	2	3	2	3
	3	3	3	3	3

XOR

screen

		0	1	2	3
array	0	0	1	2	3
	1	1	0	3	2
value	2	2	3	0	1
	3	3	2	1	0

COMMANDS, STATEMENTS AND FUNCTIONS

Animation

The GET and PUT statements allows the animation of objects, through the use of the following procedure:

1. PUT the image on the screen (with the XOR option)
2. Calculate the new position of the rectangle that contains the image.
3. PUT the image on the screen (with the XOR option) a second time at the old location to remove the old image
4. Go to step 1, but this time PUT the image at the new location.

Movement done this way will leave the background color unchanged. Flicker can be cut down by minimizing the time between steps 4 and 1, and by making sure that there is enough time delay (using a FOR/NEXT delay loop, for example) between 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background color, animation can be performed using the PSET *action-verb* statement. The idea is to leave a border around the image when it is first displayed as large or larger than the maximum distance the object will move. Thus, when an object is moved, this border will effectively erase any points left by the previous PUT. This method may be somewhat faster than the method using XOR described above, since only one PUT is required to move an object (although you must PUT a larger image).

Possible Errors

An "Illegal function call" error occurs if the image to be transferred is too large to fit on the screen.

RANDOMIZE Statement

Reseeds the random number generator.

RANDOMIZE [*numexp*]

Where

SYNTAX ELEMENT	MEANING
<i>numexp</i>	Is any numeric expression. The value of the expression will be used to seed the random numbers.

Characteristics

If *numexp* is omitted, GW-BASIC suspends program execution and asks for a value by displaying:

Random Number Seed (-32768 to 32767)?

before executing RANDOMIZE.

To get a new random seed without prompting the user, use the numeric TIMER function. For example:

RANDOMIZE TIMER

COMMANDS, STATEMENTS AND FUNCTIONS

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

Example

```
10 RANDOMIZE
20 FOR I = 1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)? 3
.2226007 .594141419 .2414202 .2013798 5.361748E-02
Ok
RUN
Random Number Seed (-32768 to 32767)? 4
.628988 765605 .5551561 .775797 .7834911
Ok
RUN
Random Number Seed (-32768 to 32767)? 3
.2226007 .594141419 .2414202 .2013798 5.361748E-02
Ok
```

READ Statement

Reads values from one or more DATA statement and assigns them to the specified variables. The READ statement is usually used in a program.

READ *variable* [, *variable*] ...

Where

SYNTAX ELEMENT	MEANING
<i>variable</i>	Each <i>variable</i> in the list may be a numeric or string <i>variable</i> . The type of the <i>variable</i> must agree with the type of the associated value in the DATA statement sequence.

Characteristics

If the data type (numeric or string) of an entry in the data sequence does not correspond to the type of the associated *variable*, a "Syntax error" will result. However any numeric data type (integer, single or double precision) may be assigned to any numeric *variable*.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in the list of variables exceeds the number of elements in the DATA statement(s), an "Out of data" error message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see "RESTORE" later in this chapter).

COMMANDS, STATEMENTS AND FUNCTIONS

Example 1

```
.  
. .  
80 FOR I=1 TO 10  
90 READ A(I)  
100 NEXT I  
110 DATA 3.08,5.19,3.12,3.98,4.24  
120 DATA 5.08,5.55,4.00,3.16,3.37  
. .  
.
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08 the value of A(2) will be 5.19 and so on.

Example 2

```
10 PRINT "CITY", "STATE", "ZIP"  
20 READ C$,S$,Z  
30 DATA "DENVER,", COLORADO, 80211  
40 PRINT C$,S$,Z  
Ok  
RUN  
CITY          STATE          ZIP  
DENVER        COLORADO      80211
```

This program READs string and numeric data from the DATA statement in line 30.

REM Statement

Allows explanatory remarks to be inserted in a program.

REM *remark*

Where

SYNTAX ELEMENT	MEANING
<i>remark</i>	Represents a sequence of characters

Characteristics

REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into from a GOTO or GOSUB statement. Execution will continue with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark (') instead of REM. The single quotation mark may also be entered just after the line number, like REM.

COMMANDS, STATEMENTS AND FUNCTIONS

Do not use remarks in a DATA statement, because it would be considered legal data.

Examples

```
.  
.   
.   
120 REM Calculate Average Velocity  
130 FOR I=1 TO 20  
140 SUM=SUM + V(I)  
150 NEXT I
```

```
.  
.   
.   
or
```

```
.  
.   
.   
120 FOR I=1 TO 20      'Calculate Average Velocity  
130 SUM=SUM + V(I)  
140 NEXT I
```

```
.  
.   
. 
```

or

```
.  
.   
.   
120 'Calculate Average Velocity  
130 FOR I=1 TO 20  
140 SUM=SUM + V(I)  
150 NEXT I
```

```
.  
.   
. 
```


RENUM Command

Changes the line numbers of the current program. RENUM is usually used in immediate mode.

RENUM [*new-linenum*][, [*old-linenum*][, *increment*]]

Where

SYNTAX ELEMENT	MEANING
<i>new-linenum</i>	Is the first line number to be used in the new sequence. The default is 10.
<i>old-linenum</i>	Is the line in the current program where renumbering is to begin. The default is the first line of the program.
<i>increment</i>	Is the increment to be used in the new sequence. The default is 10.

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, RESTORE, RESUME, and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line number xxxxx in yyyy" is printed. The incorrect line number reference xxxxx is not changed by RENUM, but line number yyyy may be changed.

RENUM cannot be used to change the order of program lines or to create line numbers greater than 65529. An "Illegal function call" error will result.

Examples

RENUM

Renumbers the entire program. The first new line number will be 10. Lines will be numbered in increments of 10.

RENUM 300,,50

Renumbers the entire program. The first new line number will be 300. Lines will be numbered in increments of 50.

RENUM 1000,900,20

Renumbers the lines from 900 up, so they start with line number 1000 and are numbered in increments of 20.

RESET Command


Closes all open data files on all drives. RESET is usually used in a program.

RESET

Characteristics

RESET closes all open data files on all drives, and forces all blocks in memory to be written to disk. Thus, if the machine loses power, all files will be properly updated. All files must be closed before a disk is removed from its drive.

Note that RESET performs the same action as CLOSE with no arguments, if all open data files are resident on disk.



RESTORE Statement

Permits DATA statements to be re-read either from the beginning of the internal data file or from a specified line. RESTORE is usually used in a program.

RESTORE [*linenum*]

Where

SYNTAX ELEMENT	MEANING
<i>linenum</i>	Must be the line number of a DATA statement

Characteristics

After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If *linenum* is specified, the next READ statement accesses the first data item in the specified DATA statement.

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 58, 67, 97
.
.
.
```

RESUME Statement

Continues program execution after an error trapping routine has been performed. RESUME is usually used in a program.

RESUME [0 | NEXT | *linenum*]

Where

SYNTAX	MEANING
RESUME or RESUME 0	Execution resumes at the statement which caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME <i>linenum</i>	Execution resumes at the specified line.

Remarks

Any one of the four formats shown above may be used, depending upon where execution is to resume.

A RESUME statement that is not in an error handling routine causes a "RESUME without error" message to be printed.

Example

```
10 ON ERROR GOTO 900
.
.
.
900 IF (ERR = 230)AND(ERL = 90) THEN PRINT "TRY
      AGAIN":RESUME 80
.
.
.
```

RIGHT\$ Function

Returns a substring from a specified string, extracting its rightmost characters.

RIGHT\$ (*string* , *length*)

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
<i>string</i>	Is a string expression whose value is the original string from which a substring is to be returned.
<i>length</i>	Is a numeric expression rounded to the nearest integer, whose value (from 0 to 255) represents the length of the returned string.

Remarks

If *length* is greater or equal to `LEN(string)`, then the entire original string is returned. When `length = 0`, the null string (length of zero) is returned.

Example

```
10 A$ = "DISK GWBASIC"  
20 PRINT RIGHT$(A$,7)  
RUN  
GWBASIC  
Ok
```

RMDIR Command

Removes an existing directory. RMDIR is usually used in immediate mode.

RMDIR *pathname*

Where

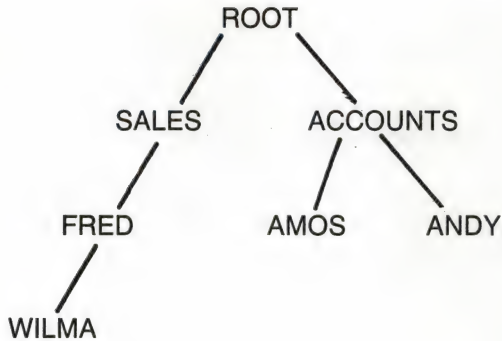
SYNTAX ELEMENT	MEANING
<i>pathname</i>	Is the name of the directory which is to be deleted

Characteristics

RMDIR works exactly like the MS-DOS command RMDIR. The directory to be deleted must be empty of all files and sub-directories except the working directory ('.') and the parent directory ('..') entries, or a "Path not found" error is given.

COMMANDS, STATEMENTS AND FUNCTIONS

Example



With reference to our sample structure above, we decide that we no longer want the sub-directory ANDY. Let us assume that our current directory is ROOT. Then:

RMDIR "ACCOUNTS\ANDY"

deletes the directory ANDY.

On the other hand, if you want to make ACCOUNTS the current directory and remove the directory called AMOS then:

CHDIR "ACCOUNTS"
RMDIR "AMOS"

Possible Errors

"Path not found"

"Bad File name"

"Path/File Access Error": usually indicating that the directory is not empty.

RND Function

Returns a random number between 0 and 1.

RND [(*numexp*)]

Where

SYNTAX ELEMENT	MEANING
<i>numexp</i>	Is a numeric expression which modifies the returned value.

Characteristics

RND returns a uniformly distributed random number in the open interval between 0 and 1. Unless you write a RANDOMIZE statement before the RND statement, the same sequence of random numbers is generated on every run.

RND acts differently depending upon whether the numeric expression evaluates to a positive number, negative number, or zero:

- RND (positive number) returns the next number in the current sequence
- RND (negative number) reseeds the random number generator and returns the first random number in the new sequence
- RND(0) returns the last random number generated, without affecting the current sequence.

COMMANDS, STATEMENTS AND FUNCTIONS

The numeric expression is optional. If you do not give one, RND acts as if you had given a positive expression as an argument.

To return integer random numbers in the range 0 (zero) to N, use:

```
INT (RND*(N + 1))
```

Example

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
 12 65 86 72 79
Ok
```

RUN Command

Runs the current program or loads a program from disk into memory and runs it.

Syntax 1

```
RUN [linenum]
```

Syntax 2

```
RUN {filespec|pathname}[ ,R ]
```

Where

SYNTAX ELEMENT	MEANING
<i>linenum</i>	Is the line number of the program resident in memory. If <i>linenum</i> is specified execution begins on that line.
<i>filespec</i> (or <i>pathname</i>)	Is a string expression which specifies the program to be loaded and run.
R	Specifies that all data files (that were opened before loading the designated program) remain open.

Characteristics

For a program currently in memory, if *linenum* is specified, execution begins on that line, otherwise, execution begins at the lowest line number. To run a program which is not resident in memory, the *filespec* or *pathname* option must be entered. This option will specify the same filename that was used when the file was **SAVE**d. (MS-DOS will append a default .BAS filename extension if one was not supplied in the **SAVE** command.)

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the **R** option, all data files remain open.

COMMANDS, STATEMENTS AND FUNCTIONS

Examples

RUN "B:NEWFILE",R

RUN A

RUN 150

RUN "C:\R001\R002"

SAVE Command

Saves the program resident in memory onto disk and gives it a name. Option **A** saves the program in ASCII format. Option **P** saves it protected.

SAVE {*filespec*|*pathname*} [, {**A** | **P** }]

Where

SYNTAX ELEMENT	MEANING
<i>filespec</i> (or <i>pathname</i>)	Is a string expression which specifies the name of the file to be saved, and optionally the drive. If the filename extension is omitted, .BAS is assumed. If the drive is omitted, the default MS-DOS drive is assumed.
A	Saves the program file in ASCII format. If the A option is not specified, GW-BASIC saves the file in a compressed binary format.

SYNTAX ELEMENT	MEANING
P	Protects the file by saving it in an encoded binary format. When a protected file is later run (or LOADED), any attempt to list or edit it will fail.

Characteristics

If a file with the same name already exists on the selected disk, it will be written over.

GW-BASIC saves the file in a compressed binary format, unless the A option is specified.

ASCII format takes more space on disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some MS-DOS commands such as TYPE may require an ASCII format file.

Attempts to MERGE binary programs will result in a "Bad file mode" error.

Examples

SAVE "SUPERB"

Saves the program in memory on the default drive as SUPERB.BAS.

SAVE "A:PROG",A

Saves in ASCII the program in memory on the diskette inserted on drive A, as PROG.BAS.

SAVE "B:SECRET",P

Saves protected the program in memory on the diskette inserted on drive B, as SECRET.BAS.

SCREEN Function

Returns either the ASCII code (0-255) or the color number for the character at the specified screen location.

SCREEN(*row* , *column*[, *condition*])

Where

SYNTAX ELEMENT	MEANING
<i>row</i>	Is a numeric expression returning an unsigned integer in the range 1 to 25
<i>column</i>	Is a numeric expression returning an unsigned integer the range of which depends on the screen width
<i>condition</i>	Is a valid numeric, relational or logical expression returning a boolean result (0 or 1). If <i>condition</i> is given as non-zero, the color number for the character is returned instead of the ASCII code.

Characteristics

The SCREEN function returns zero if the system is in one of the graphics modes and the specified screen location contains graphics data.

Refer to Appendix C for a complete list of ASCII codes.


Examples

100 X = SCREEN (10,10) 'If the character at 10,10 is A then 65 is returned.

110 X = SCREEN (1,1,1) 'Returns the color number of the character in the upper left hand corner of the screen.

Errors

If you enter a value outside the above mentioned ranges, an "Illegal function call" error is returned.



SCREEN Statement

Allows you to pass from Text Mode to one of the graphics modes. It also allows you to enable/disable color and to select the active and visual page (in Text Mode).

SCREEN [*mode*] [, [*burst*] [, [*apage*] [, *vpage*]]]

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
<i>mode</i>	Is a numeric expression resulting in an integer value in the range 0 to 255. It defines either Text Mode (0), Medium-Resolution Graphics Mode (1), High-Resolution Graphics Mode (2), or Super-Resolution Graphics Mode (3 to 255).
<i>burst</i>	<p>Is a numeric expression resulting in an integer value of 0 or 1. It enables color on a non-standard monitor. In Text Mode a 0 value disables color, and a 1 value enables color. In Medium Resolution a 0 value enables color, and a 1 value disables color. Both in High Resolution and Super Resolution the <i>burst</i> value is ignored, as these two modes only support monochrome.</p> <p>For a standard monitor, this parameter has no meaning.</p>
<i>apage</i> (Text Mode only)	Is an integer expression in the range 0 to 7 for width 40, or 0 to 3 for width 80. It selects the active page, i.e. the page to be written to by output statements to the screen. If omitted, the active page defaults to 0.
<i>vpage</i> (Text Mode only)	Is an integer expression in the range 0 to 7 for width 40, or 0 to 3 for width 80. It selects the visual page, i.e. the page to be displayed on the screen which may be different from the active page. If you omit this parameter the visual page will default to the active page.

mode and burst Parameters

In the following table the first two columns are the *mode* and *burst* parameters of a SCREEN statement.

The *burst* parameter enables color on non-standard color monitors. For systems with standard monitors, this parameter has no real meaning. For example a *burst* value of 0 or 1 in Medium Resolution will have the same effect if a standard color monitor is used; likewise it will have the same effect if a standard monochrome monitor is used (in this case the four colors will appear as shades of grey).

mode	burst	Description
0	0	80 c. x 25 r. - B/W Text Mode
0	1	80 c. x 25 r. - Color Text Mode
1	0	320 hor.pixels x 200 vert.pixels-Color Medium Resolution Graphics Mode (40 c. x 25 r.)
1	1	320 hor.pixels x 200 vert.pixels-B/W Medium Resolution Graphics Mode (40 c. x 25 r.)
2	x (ignored)	640 hor.pixels x 200 vert.pixels-B/W High Resolution Graphics Mode (80 c. x 25 r.)
3-255	x (ignored)	640 hor.pixels x 400 vert.pixels B/W Super Resolution Graphics Mode (80 c. x 25 r.)

Tab. 8-3 Use of mode and burst Parameters

COMMANDS, STATEMENTS AND FUNCTIONS

Default Values

If you do not enter a SCREEN statement, the system assumes the following default values:

mode = 0 (Text Mode)
burst = 0 (B/W)
apage = 0 (active page 0)
vpage = 0 (virtual page 0)

The SCREEN statement must precede any I/O statement to the screen, that uses attributes different to those currently in force. You can use more than one SCREEN statement to define different screen attributes for different sections of your program.

Active and Visual Pages

If Text Mode is selected, you can specify two more parameters (*apage* and *vpage*) to select the active and visual page. There are eight display pages (numbered 0 to 7) in 40-column Text Mode, and four display pages (numbered 0 to 3) in 80-column Text Mode. Only one display page is available in any of the three graphics modes.

Only one cursor is shared between the pages, thus, if you select a new active page, you must save the cursor position (by POS(0) and CSRLIN) before changing to the new page. If you return to the original active page, you must restore the cursor position by the LOCATE (Text) statement. If you use the SCREEN statement only to change the pages, you can omit the first two parameters (*mode* and *burst*).

Screen Width

At initialization the screen width is 80 columns, thus you should use the WIDTH statement to select a 40-column screen. If you select the Medium Resolution by the SCREEN statement, this also causes the number of columns to be 40 (without using the WIDTH statement).

In Text Mode, the WIDTH statement may be used to select between the 40-column mode and the 80-column mode. Likewise, the WIDTH statement may be used to select between modes 1 and 2 (Medium or High resolution mode).

Selecting Text Mode (*mode* = 0) after selection of one of the graphics modes will select either a 40-column screen or an 80-column screen, depending on the width used in the graphics mode. For example:

SCREEN 1 Set Screen to Medium Res. Mode (WIDTH = 40)
SCREEN 0 Changes Screen to 40x25 Text Mode (WIDTH = 40)

Remarks

IF...	THEN...
all parameters are valid	the new screen mode is saved, the screen is erased, the foreground and the background colors are set to their default values.
all parameters are identical to the preceding ones	nothing is altered.
you omit a parameter	it assumes the preceding value (except for the visual page that defaults to the active page).

COMMANDS, STATEMENTS AND FUNCTIONS

Examples

10 SCREEN 0,1,0,0
Select Text mode with color,
Active and visual page to 0.

20 SCREEN ,,1,2
mode and *burst* unchanged,
use active page 1,
display page 2.

30 SCREEN 2
Switch to High Res. Graphics Mode.

40 SCREEN 1,1
Switch to Medium Res. Color Graphics.

50 SCREEN ,0
Medium Res. Graphics, color off.

Possible Errors

If you enter a value outside the specified ranges, an "Illegal function call" error is returned.

SGN Function

Returns a value determined by the sign of the specified numeric expression.

SGN(*numexp*)

Characteristics

If *numexp* > 0, SGN(*numexp*) returns 1.
If *numexp* = 0, SGN(*numexp*) returns 0.
If *numexp* < 0, SGN(*numexp*) returns -1.

Example

```
50 ON SGN(X) + 2 GOTO 300,400,500
```

branches to 300 if *numexp* is negative, 400 if *numexp* is 0, and 500 if *numexp* is positive.

SHELL Command

Loads into memory and executes another program (.EXE or .COM or .BAT).

SHELL [*stringexp*]

Where

SYNTAX ELEMENT	MEANING
<i>stringexp</i>	Is a string expression containing the name of a program to run and (optionally) command arguments

Characteristics

When the program finishes, control returns to the GW-BASIC program at the statement following the SHELL command. A program executed under control of GW-BASIC is referred to as a "Child process".

COMMANDS, STATEMENTS AND FUNCTIONS

Child processes (or "children") are executed by SHELL loading and running a copy of COMMAND with the /C switch. By using COMMAND this way command line parameters are passed to the child. Standard Input and Output may be selected, and Built-in Commands such as DIR, PATH, and SORT may be executed.

Remarks

The program name in *stringexp* may have any extension you want since COMMAND has to worry about it. If no extension is supplied, COMMAND will look for a .COM file, then a .EXE file, and finally, a .BAT file. If COMMAND is not found, SHELL will issue a "File not found" error. No GW-BASIC error is generated if COMMAND cannot find the file specified in *stringexp*.

Any text in *stringexp* separated from the program name by at least 1 blank, will be processed by COMMAND as a sequence of program parameters.

GW-BASIC remains in memory while the child process is running. When the child finishes, GW-BASIC continues:

SHELL with no *stringexp* will give you a new COMMAND shell. You may now do anything that COMMAND allows. When ready to return to GW-BASIC, enter the MS-DOS command: EXIT.

Examples

SHELL	(get a new COMMAND)
A> DIR	(user enters DIR to see files)
A> EXIT	(user enters EXIT to return to GW-BASIC)

The following example writes some data to be sorted, uses SHELL to execute SORT to sort it, then reads the sorted data to write a report.

```
900 OPEN "SORTIN.DAT" FOR OUTPUT AS #1
...
950 REM write data to be sorted
...
1000 CLOSE 1
```

```

1010 SHELL "SORT <SORTIN.DAT >SORTOUT.DAT"
1020 OPEN "SORTOUT.DAT" FOR INPUT AS #1
1030 REM Process the sorted data
...
10 SHELL "DIR | SORT >FILES.
20 OPEN "FILES." FOR INPUT AS #1
...

```

Possible Errors

"File not found": if COMMAND could not be found.

"Out of memory": there was not enough memory to run the Child.

"Can't continue after SHELL": there is not enough memory for GW-BASIC to continue. All files are closed and GW-BASIC returns to MS-DOS. This may happen when a Child process "terminates and stays resident".

"Internal error": either GW-BASIC or MS-DOS is not functioning correctly.

SIN Function

Calculates the sine of the argument.

SIN(*numexp*)

Characteristics

The SIN function is calculated in single precision, unless /D is supplied in the GWBASIC command line.

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
PRINT SIN(1.5)
.9974951
Ok
```

SOUND Statement

Produces sound.

SOUND *frequency* , *duration*

Where

SYNTAX ELEMENT	MEANING
<i>frequency</i>	Is a numeric expression from 37 to 32767. It represents the frequency in Hertz
<i>duration</i>	Is the duration in clock ticks. Clock ticks occur 18.2 times per second. Duration is an integer expression from 0 to 65535.

Characteristics

If the duration is zero, any SOUND statement that is running will be turned off. If no SOUND statement is currently running, a SOUND statement with a duration of zero will have no effect.

This following table displays the frequencies of musical notes (two octaves below and two octaves above middle C).

1975.5 B	1760.0 A	1568.0 G	1396.9 F	1318.5 E	1174.7 D	1046.5 C
987.77 B	880.00 A	783.99 G	698.46 F	659.26 E	587.33 D	523.25 C
493.88 B	440.00 A	392.00 G	349.23 F	329.63 E	293.66 D	261.63 C
246.94 B	220.00 A	196.00 G	174.61 F	164.81 E	146.83 D	130.81 C

COMMANDS, STATEMENTS AND FUNCTIONS

Tempos and Beats/Minute

TEMPOS	BEATS/MINUTE	TICKS/BEAT
Larghissimo Largo Larghetto Grave Lento Adagio	40-60 60-66 66-76	28.13-18.75 18.75-17.05 17.05-14.8
Adagietto Andante	76-108	14.8-10.42
Andantino Moderato	108-120	10.42-9.38
Allegretto Allegro Vivace Veloce Presto	120-168 168-208	9.38-6.7 6.7-5.41
Prestissimo	greater than 208	less than 5.41

Tab. 8-4 Tempos and Beats/Minute

Example

100 SOUND RND*1000 + 37,2

This statement creates random sounds.

SPACES\$ Function

Returns a string of a specified number of spaces.

SPACES\$(*length*)

Where

SYNTAX ELEMENT	MEANING
<i>length</i>	Is an integer expression from 0 to 255. It specifies the number of spaces i.e. the length of the returned string.

Example

```
10 FOR I=1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
1
2
3
4
5
Ok
```

COMMANDS, STATEMENTS AND FUNCTIONS

Possible Errors

If *length* is outside the specified range, an "Illegal function call" error is returned.

SPC Function

Skips spaces in a PRINT, LPRINT, or PRINT # statement.

SPC(*n*)

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is an integer expression from 0 to 255. It specifies the number of spaces to be inserted in the output line.

Remarks

SPC may only be used with PRINT, LPRINT and PRINT # statements.

If *n* is greater than the defined width, then the value used is *n* MOD *width*. A semicolon (;) is assumed to follow the SPC function; thus GW-BASIC does not add a carriage return, if you enter an SPC function at the end of a list of data.

If *n* is outside the specified range an "Illegal function call" error is returned.

Example

```
PRINT "FOUR" SPC(15) "SEASONS"  
FOUR           SEASONS  
Ok
```



SQR Function

Returns the square root of a positive numeric expression.

SQR(*numexp*)

Remarks

SQR is calculated in single precision, unless the /D switch is supplied in the GWBASIC command line.

An "Illegal function call" error results if the argument is negative.

Example

```
10 FOR X= 10 TO 25 STEP 5  
20 PRINT X, SQR(X)  
30 NEXT  
RUN  
10           3.162278  
15           3.872984  
20           4.472136  
25           5  
Ok
```

STOP Statement

Interrupts program execution then returns to command level.

STOP is only used in a program.

STOP

Characteristics

A STOP statement may be used anywhere in a program. When a STOP is encountered, the following message is displayed:

Break in *nnnnn*

The STOP statement does not close files, unlike the END statement.

GW-BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command.

Example

```
10 INPUT A,B,C
20 K = A ^ 2 * 5.3 : L = B ^ 3 / .26
30 STOP
40 M = C * K + 100 : PRINT M
RUN
? 1,2,3
Break in 30
Ok
```

```
PRINT L
 30.76923
Ok
CONT
 115.9
Ok
```



STR\$ Function

Returns the string representation of the value of a specified numeric expression.

STR\$(*numexp*)

Example

```
10 A$ = STR$(70)
20 PRINT A$
Ok
RUN
 70
Ok
```

70 (the argument of STR\$) is a number, but the contents of A\$ is a three character string whose value is 70. The first character holds the sign of the number; a blank for positive, a minus sign for negative.

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
.
.
.
```

The entered number N is converted to a string by the STR\$ function.

Example

```
10 A = 1.3
20 A = VAL(STR$(A!))
30 PRINT A
Ok
RUN
1.3
Ok
```

VAL is the complementary function of STR\$.

STRING\$ Function

Returns a string of specified length whose characters all have the same ASCII code or equal the first character of a given string.

Syntax 1

STRING\$(length , code)

Syntax 2

STRING\$(*length* , *stringexp*)

Where

SYNTAX ELEMENT	MEANING
<i>length</i>	Is an integer expression in the range 0 to 255. It specifies the length of the resulting string.
<i>code</i>	Is an integer expression in the range 0 to 255. It specifies the ASCII code whose corresponding character is used to form the resulting string.
<i>stringexp</i>	Is a string expression whose first character is used to form the resulting string.

Example

```
10 X$ = STRING$(10,45)
20 PRINT X$,"MONTHLY REPORT";X$
RUN
-----MONTHLY REPORT-----
Ok
```

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
10 LET A$ = "HOUSTON"  
20 LET X$ = STRING$(8,A$)  
30 PRINT X$  
RUN  
HHHHHHHH
```

SWAP Statement

Exchanges the values of two variables.

SWAP *variable1* , *variable2*

Where

SYNTAX ELEMENT	MEANING
<i>variable1</i> and <i>variable2</i>	Are two variables of the same type (integer, single-precision, double-precision, or string)

Remarks

The two variables to be exchanged must be of the same type or a "Type mismatch" error occurs. If the second variable is not already defined when SWAP is executed, an "Illegal function call" error will result.

Example

```
Ok
10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
  ONE FOR ALL
  ALL FOR ONE
Ok
```

After line 30 is executed, A\$ has the value " ALL " and B\$ has the value " ONE ".



SYSTEM Command

Closes all open data files and returns control to MS-DOS.

SYSTEM

Characteristics

When a SYSTEM command is executed, all open files are closed, the current program is lost, and control is returned to MS-DOS. If GW-BASIC has been entered through a Batch file from MS-DOS, SYSTEM returns control to the Batch file.

TAB Function

Tabs the cursor or the print head to a specified position in PRINT, LPRINT, or PRINT # statements.

TAB(*n*)

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is an integer expression from 1 to 255

Characteristics

If the current cursor or print position is already beyond the specified value *n*, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one.

If the value of *n* exceeds the defined width, the modulo operation is applied. For example PRINT TAB(243) on a 40-column screen is the same as PRINT TAB(3), because $243 \text{ MOD } 40 = 3$.

A semicolon is assumed to follow the TAB function, thus GW-BASIC does not add a carriage return if you enter a TAB function at the end of a list of data.

Example

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES", "$25.00"
```

RUN

NAME	AMOUNT
------	--------

G. T. JONES	\$25.00
-------------	---------

Ok

TAN Function

Returns the tangent of the argument.

TAN(*numexp*)

Where

SYNTAX ELEMENT	MEANING
<i>numexp</i>	Is a numeric expression representing the angle in radians

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

TAN(*numexp*) is calculated in single precision (unless /D is supplied in the GWBASIC command line).

If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example

```
PRINT TAN (3.14/4)
.999204
Ok
```

TIME\$ Statement and Function

The TIME\$ statement sets the current time.

The TIME\$ function retrieves the current time.

Syntax 1: statement

TIME\$ = *stringexp*

Syntax 2: function

stringvar = **TIME\$**

Where

SYNTAX ELEMENT	MEANING
<i>stringexp</i>	Is a string expression indicating the time to be set
<i>stringvar</i>	Is a string variable in which the current time (8 character string) is returned

Characteristics

As a statement (to set the time):

stringexp is a string expression indicating the time in one of the following forms:

hh (sets the hour; minutes and seconds default to 00)

hh:mm (sets the hour and minutes; seconds default to 00)

hh:mm:ss (sets the hour, minutes and seconds)

A 24 hour clock is used; therefore 8:00 p.m. would be entered as 20:00:00.

You may omit a leading zero to specify the values of hours, minutes and seconds, but you must enter at least one digit. The time may also have been set by MS-DOS prior to entering GW-BASIC.

As a function (to retrieve the time):

The TIME\$ function returns an eight character string in the form *hh:mm:ss*, where *hh* is the hour (00 through 23), *mm* is minutes (00 through 59), and *ss* is seconds (00 through 59).

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
TIME$ = "8:0"  
Ok  
PRINT TIME$  
08:00:04  
Ok
```

Example


The following program displays the current date and time on the 25th line of the screen and will "chime" on the hour.

```
10 KEY OFF:SCREEN 0:WIDTH 40:CLS  
20 LOCATE 25  
30 PRINT DATE$,TIME$  
40 SEC = VAL(MID$(TIME$,7,2))  
50 IF SEC = SSEC THEN 20 ELSE SSEC = SEC  
60 IF SEC = 0 THEN 1010  
70 IF SEC = 30 THEN 1020  
80 IF SEC < 57 THEN 20  
  
1000 SOUND 1000,2:GOTO 20  
1010 SOUND 2000,8:GOTO 20  
1020 SOUND 400,4 :GOTO 20
```

Possible Errors

An "Illegal function call" error is returned, and the previous time is retained, if you enter a value outside the corresponding range.

A "Type mismatch" error is returned, if you do not enter a valid string for *stringexp* .



TIMER Function

Returns a single-precision number indicating the seconds that have elapsed since midnight or system reset.

TIMER

Characteristics

TIMER is a numeric function. It calculates fractional seconds to the nearest degree possible. It may not be used as a user variable.

Example

```
10 FOR K = 1 TO 10
20 PRINT "TIMER = ";TIMER
30 NEXT
```



TIMER Statements

TIMER ON enables TIMER event trapping.
TIMER OFF disables TIMER event trapping.
TIMER STOP suspends TIMER event trapping.

TIMER {ON|OFF|STOP}

COMMANDS, STATEMENTS AND FUNCTIONS

Remarks

The **TIMER ON** statement enables real time event trapping by an **ON TIMER(n) GOSUB** statement. While trapping is enabled, GW-BASIC checks between every statement to see if the timer has reached the specified level. If it has, the **ON TIMER(n) GOSUB** statement is executed.

TIMER OFF disables the event trap. If an event takes place, it is not remembered if a subsequent **TIMER ON** is used.

TIMER STOP disables the event trap, but if an event occurs, it is remembered and an **ON TIMER(n) GOSUB** statement will be executed as soon as trapping is enabled.

When an **ON TIMER(n) GOSUB** is performed, an automatic **TIMER STOP** is executed. Once an error trap takes place, all trapping is automatically disabled.

Also see **ON TIMER(n) GOSUB** statement in this chapter.

TRON/TROFF Commands

TRON (**TRACE ON**) causes the line number of each statement executed to be listed.

TROFF (**TRACE OFF**) stops the line number listing initiated by **TRON**.

Syntax 1

TRON

Syntax 2

TROFF

Characteristics

The TRON command (executed in either immediate or program mode) is used as a debugging tool. With TRON in operation, each line number of the program is displayed on the screen as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF command (or when a NEW command is executed).

Example

TRON

Ok

LIST

10 K = 10

20 FOR J = 1 TO 2

30 L = K + 10

40 PRINT J;K;L

50 K = K + 10

60 NEXT

70 END

Ok

RUN

[10][20][30][40] 1 10 20

[50][60][30][40] 2 20 30

[50][60][70]

Ok

TROFF

Ok

COMMANDS, STATEMENTS AND FUNCTIONS

UNLOCK Statement

The UNLOCK statement releases locks applied to a opened file. This statement is only of use if MS-NET is installed with MS-DOS release 3.1 or later.

UNLOCK [#] *filenum* [, *recordnum1*] [**TO** *recordnum2*]

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is the file number of the opened file.
<i>recordnum1</i>	Is the number of the record at which unlocking begins. If this parameter is omitted record number 1 is assumed.
<i>recordnum2</i>	Is the number of the record at which unlocking terminates. If this parameter is omitted, only one record <i>recordnum1</i> is unlocked.

Characteristics

If a record number or range of record numbers is specified, and the file is opened in random mode, only those records in the range are unlocked. The record number range must exactly match the record number range given in the LOCK statement or a "Permission denied" error message will be returned.

See also the LOCK and OPEN statements.

Note

If you try and use this statement with an MS-DOS release prior to 3.1 you will get an "Advanced Feature Error". If you are using MS-DOS 3.1 or later, but MS-NET is not installed, you will get a "Permission Denied" error. The MS-DOS command SHARE must also be given (either at the MS-DOS prompt or from an AUTOEXEC.BAT file). See the "MS-DOS User Guide" for a full description of procedures to be followed for networking.

The suggested usage of files on shared devices is for a LOCK to be executed on a file, or record range within a file, before an attempt is made to read or write to that file. It is also recommended that the file or range be UNLOCKed before the file is closed (failure to do so may cause problems for future access to that file in a network environment). The time in which files are locked should be kept to minimum.

Examples

```
LOCK #1, 1 TO 4
LOCK #1, 5 TO 8
.
.
.
UNLOCK #1, 1 TO 4
UNLOCK #1, 5 TO 8
```

Note that the UNLOCK statements in the above example cannot be replaced by the single statement:

```
UNLOCK #1, 1 TO 8
```

Possible Errors

"Permission Denied": MS-NET not installed or no preceding matching LOCK statement.

USR Function

Calls a machine language subroutine. USR is usually used in a program.

USR [*n*] (*argument*)

Where

SYNTAX ELEMENT	MEANING
<i>n</i>	Is an integer from 0 to 9. It specifies which USR subroutine is being called. If omitted USR0 is assumed.
<i>argument</i>	Is the value passed to the subroutine. It may be any numeric or string expression. Even if the subroutine does not require an argument, a dummy argument must be supplied.

Characteristics

The type (numeric or string) of the variable receiving the function call must be consistent with that of the argument passed.

Prior to calling each USR function, a corresponding DEF USR statement must be executed to define the offset of the subroutine with respect to the start address of the segment, specified in the last DEF SEG statement executed.

If no DEF SEG has been executed the start address of the default segment (the GW-BASIC Data Segment) will be used.

The CALL statement is another way to call a machine language subroutine.

Example

```
100 DEF SEG = &H8000
110 DEF USR0 = 0
120 X = 5
130 Y = USR0(X)
140 PRINT Y
```



VAL Function

Converts the string representation of a number to its numeric value.

VAL(*stringexp*)

Characteristics

The VAL function strips leading blanks, tabs, and linefeeds from the argument string.

The remaining string is converted to a number, if it is a valid numeric representation, otherwise VAL returns 0 (zero). For example:

VAL(" -3")

returns -3.

VAL("ABC")

returns 0

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699
   THEN PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815
   THEN PRINT NAME$ TAB(25) "LONG BEACH"
```

VARPTR Function

VARPTR (*variable*) returns the memory address of *variable*.
VARPTR (*# filenum*) returns the starting address of the File Control Block (FCB) associated with the file specified in *filenum*.

Syntax 1

VARPTR (*variable*)

Syntax 2

VARPTR (*# filenum*)

Where

SYNTAX ELEMENT	MEANING
<i>variable</i>	Is any numeric or string program variable
<i>filenum</i>	Is the number assigned to the file when it was opened

Remarks

For both syntax 1 and 2, the address returned will be an integer in the range -32768 to 65535. This integer value is the offset into GW-BASIC's Data Segment. If a negative address is returned, add it to 65536 to obtain the actual address.

Syntax 1

Returns the address of the first byte of data identified with *variable*.

The *variable* must have been defined prior to execution of VARPTR. Otherwise an "Illegal function call" error results. Variables are defined by executing any reference to the variable. Both numeric and string variables may be used. For string variables, the address of the first byte of the string description is returned (see Appendix F).

VARPTR is usually used to obtain the address of a variable or array so that it may be passed to a machine language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

COMMANDS, STATEMENTS AND FUNCTIONS

All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

Syntax 2

For sequential files, VARPTR (*# filenum*) returns the starting address of the File Control Block (FCB)

Example

```
10 X = USR(VARPTR(Y))
```

Example

```
100 FIELDADDRESS = VARPTR( # 2)
```

VARPTR\$ Function

Returns the memory address of the variable in string format.

VARPTR\$ (*variable*)

Characteristics

VARPTR\$ is primarily used to execute substrings with the DRAW and PLAY statements in programs that will later be compiled. With programs that will not be later compiled, the standard syntax of the DRAW and PLAY statements will be sufficient to produce the desired effects.

For example:

```
PLAY "XA$;"
```

and

```
PLAY "X" + VARPTR$(A$)
```

produce the same effect.

The *variable* must have been defined prior to the execution of the VARPTR\$ function, otherwise, an "Illegal function call" error results. Variables are defined by executing any reference to the variable. Both numeric and string variables may be used.

VARPTR\$ returns a three-byte string in the form:

- byte 0 = type of the variable
- byte 1 = low byte of address
- byte 2 = high byte of address

Note that type specifies the type of the variable, as follows:

- 2 integer
- 3 string
- 4 single-precision
- 8 double-precision

Because array addresses, string addresses and file data block change whenever a new variable is defined, it is unsafe to save the result of a VARPTR\$ function in a variable. It is recommended that VARPTR\$ is executed before each use of the result.



VIEW Statement

Defines the dimensions and position of the viewport for graphics activity. (Graphics Mode only.)

```
VIEW [[ SCREEN ][( vx1 , vy1 )-( vx2 , vy2 ) [ , [ color ] [ , [ border ] ] ] ]
```

COMMANDS, STATEMENTS AND FUNCTIONS

Where

SYNTAX ELEMENT	MEANING
$(vx1,vy1)(vx2,vy2)$	$(vx1,vy1)$ are the upper-left, and $(vx2,vy2)$ the lower-right coordinates of the viewport defined.
<i>color</i>	Permits the viewport to be filled with a specified color. If <i>color</i> is omitted then the viewport is not filled-in.
<i>border</i>	Permits the drawing of a border-line surrounding the viewport (if the necessary space for a border is available). If <i>border</i> is omitted, no border-line is drawn.

Characteristics

VIEW defines a rectangular area on the screen, the viewport, in which graphics are displayed, by specifying the top-left coordinates $(vx1,vy1)$ and the bottom-right coordinates $(vx2,vy2)$.

A VIEW statement without parameters defines the whole screen as the viewport.

SCREEN Option

The SCREEN option specifies that the *x* and *y* coordinates of all points are to be taken absolute to the screen (the origin is the top-lefthand corner of the screen); but only the points within the viewport will be displayed.

Using the format:

VIEW (vx1,vy1)-(vx2,vy2)

the coordinates, x and y, of all points will be taken as relative to the viewport. Therefore, given the following statements:

VIEW (10,10)-(200,100)
PSET (0,0) ,3

the point specified in the PSET statement will actually be positioned at (10,10) on the screen. Likewise, if a WINDOW statement is executed, all the points inside the window will be projected onto the corresponding points inside the viewport defined by VIEW.

Using the format:

VIEW SCREEN (vx1,vy1)-(vx2,vy2)

the coordinates, x and y, of all points will be taken as absolute. Therefore, given the following statements:

VIEW SCREEN (10,10)-(200,100)
PSET (0,0) ,3

the point specified in the PSET statement will not be displayed, because the point (0,0) falls outside of the area of the viewport; the statement PSET (10,10) ,3 would result in the point at the top-lefthand corner of the viewport being displayed. If a WINDOW statement is executed, all the points inside the window will be projected onto the corresponding points on the screen, but only those points within the viewport, defined by VIEW SCREEN, would be displayed.

Multiple Viewports

Any number of VIEW statements can be executed. Each execution of the VIEW statement results in the definition of a viewport that is then the current viewport. In order to change the current viewport, you must execute another VIEW statement.

COMMANDS, STATEMENTS AND FUNCTIONS

Viewports can be completely or partially overlapped. In order to erase all previously defined viewports, you must use a VIEW statement without parameters (defining the whole screen as the viewport) and then execute a CLS statement to clear the contents of the current viewport.

Examples

See the examples given with the WINDOW statement.

VIEW PRINT Statement

Defines a text window.

VIEW PRINT [*line1* TO *line2*]

Where

SYNTAX ELEMENT	MEANING
<i>line1</i>	Is the top line of the text window
<i>line2</i>	Is the bottom line of the text window

Characteristics

Statements and functions which operate within the text window include CLS, LOCATE, and the SCREEN function. The Screen Editor will limit functions such as scroll and cursor movement to the text window.

If no parameters are specified, VIEW PRINT will initialize the text window to include the whole screen.

Example

VIEW PRINT 1 TO 5

creates a text window of 5 lines at the top of the screen.

WAIT Statement

Suspends program execution while monitoring the status of a machine input port. WAIT may only be used in a program.

WAIT *port* , *i* [, *j*]

Where

SYNTAX ELEMENT	MEANING
<i>port</i>	Represents a port number, i.e., an integer from 0 to 65535
<i>i,j</i>	Are integer expressions from 0 to 255

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is XORed with the integer expression j , and then ANDed with i . If the result is zero, GW-BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If j is omitted, it is assumed to be zero.

It is possible to enter an infinite loop with the WAIT statement.

Example

```
100 WAIT 32,2
```

WHILE...WEND Statements

Execute a series of statements in a loop as long as a given condition remains true.

```
WHILE condition
```

```
  .
```

```
  .
```

```
    [loop statements]
```

```
  .
```

```
  .
```

```
WEND
```

Where

SYNTAX ELEMENT	MEANING
<i>condition</i>	Is a numeric, relational or logical expression. GW-BASIC determines whether the <i>condition</i> is true or false by testing the result of the expression for non zero and zero, respectively. A non zero result is true and a zero result is false. Because of this, you can test whether the value of a variable is non zero or zero by merely specifying the name of the variable as a <i>condition</i> .
<i>loop statements</i>	Are executed until a WEND statement is encountered

Characteristics

If *condition* is not zero (i.e., true), *loop statements* are executed until the WEND statement is encountered. GW-BASIC then returns to the WHILE statement and checks *condition*. If it is still not zero, the process is repeated. If it is zero (i.e. false), execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Do not direct program flow into a WHILE/WEND loop without entering through the WHILE statement.

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
90 'BUBBLE SORT ARRAY A$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115     FLIPS=0
120     FOR I=1 TO J-1
130         IF A$(I)>A$(I+1) THEN
            SWAP A$(I),A$(I+1):FLIPS=1
140     NEXT I
150 WEND
```

WIDTH Statement

Sets the line width in characters. GW-BASIC adds a carriage return after outputting the specified number of characters.

In any of the Graphics Modes it defines a text window.

Syntax 1

WIDTH [LPRINT]*size*

Syntax 2

WIDTH # *filenum* , *size*

Syntax 3

WIDTH *device* , *size*

Where

SYNTAX ELEMENT	MEANING
<i>size</i>	Is an integer expression in the range 0 to 255. It specifies the new width.
<i>filenum</i>	Is the number under which the file was opened.
<i>device</i>	Is a string expression indicating the device that is to be used. Valid devices are: SCRN:, LPT1:, LPT2:, LPT3:, COM1:, COM2:.

WIDTH LPRINT *size*

Sets the line width at the line printer.

WIDTH *size* or **WIDTH "SCRN:"**, *size*

Sets the screen width (in Text mode), selects a text window or changes mode (in Graphics mode). Changing the screen or text window width, or the mode, causes the screen to be cleared.

In Text Mode *size* may only have the values 40 or 80, selecting either a 40-column or an 80-column screen.

COMMANDS, STATEMENTS AND FUNCTIONS

In one of the graphics modes you can either change mode or select a text window to the left of the screen of width less than or equal to 40 (Medium Resolution) or less than or equal to 80 (High or Super Resolution). The height of the window is fixed as the height of the screen. The width of the function key display will correspond to the selected width. If the number of columns displayed is less than 80 columns, a **CTRL T** may be entered to scroll the function key display horizontally.

The following summarizes all possible cases.

IF <i>mode</i> is...	AND <i>size</i> is...	THEN you...
0 (Text)	40	select a 40-column screen
	80	select an 80-column screen
1 (Medium Res)	80	place the system in High Resolution (mode 2)
	$8 \leq \text{size} \leq 40$	create a text window of width <i>size</i>
2 (High Res)	40	place the system in Medium Resolution (mode 1) with 'burst' in whatever state the system was when a Text or Medium Resolution mode was last used
	$8 \leq \text{size} \leq 39$ or $41 \leq \text{size} \leq 80$	create a text window of width <i>size</i>
	size = 4	create a text window of width 40

IF <i>mode</i> is...	AND <i>size</i> is...	THEN you...
3-255 (Super Res)	$8 \leq \text{size} \leq 80$	create a text window of width <i>size</i> .
	$8 \leq \text{size} - 80 \leq 80$	create a text window of width <i>size</i> .

WIDTH #*filenum*,*size*

If the file is open, the width is immediately changed to the specified *size*. This allows the width to be changed while the file is open.

WIDTH *device*,*size*

The default line width for the specified *device* is set to *size*. The line widths of currently open files are not modified.

Stores the new *size* without changing the current width, if the device is already open. A subsequent OPEN device FOR OUTPUT AS # *n* will use the specified value for width initially.

Remarks

When the WIDTH statement causes a change in the screen mode, colors are set to their default values.

You should turn the function key display off when changing the window width (by a KEY OFF statement), otherwise, if the width is decreased, part of the old (wider) function key display may be left on the screen.

If *size* is 255, the line width is "infinite"; that is, GW-BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255. WIDTH 255 is the default for communications files.

COMMANDS, STATEMENTS AND FUNCTIONS

If you alter the width for a communications file, you do not modify the receive or the transmit buffer: GW-BASIC will insert a **CR** after a number of characters, equal to the specified *size* , has been received or sent.

Example

```
10 PRINT "ABCDEFGHJKLMNOPQRSTUVWXYZ"  
RUN  
ABCDEFGHJKLMNOPQRSTUVWXYZ  
Ok  
WIDTH 18  
Ok  
RUN  
ABCDEFGHJKLMNOPQR  
STUVWXYZ  
Ok
```

Example

```
10 WIDTH "LPT1:", 5  
20 OPEN "LPT1:" FOR OUTPUT AS 1  
30 PRINT #1, "1234567890"  
35 PRINT #1  
40 WIDTH #1, 6  
50 PRINT #1, "1234567890"  
RUN
```

will yield on the printer:

```
12345  
67890
```

```
123456  
7890
```

Example

SCREEN 1,0 'Set Screen to Medium Res. Color Graphics.

WIDTH 80 'Change Screen to High Res. Graphics.

WIDTH 40 'Changes Screen back to Medium Res.

SCREEN 0,1 'Changes Screen to 40x25 Text Color Mode.

WIDTH 80 'Changes Screen to 80x25 Text Color Mode.

Possible Errors

If *size* is outside the above specified ranges, an "Illegal function call" error is returned. The previous value is retained.

WINDOW Statement

Defines the dimensions of the current window. (Graphics Mode only).

WINDOW [**SCREEN**] (*wx1* , *wy1*)-(*wx2* , *wy2*)]

Where

SYNTAX ELEMENT	MEANING
(<i>wx1</i> , <i>wy1</i>) -(<i>wx2</i> , <i>wy2</i>)	(<i>wx1</i> , <i>wy1</i>) represent the lower-left, and (<i>wx2</i> , <i>wy2</i>) the upper-right coordinates of the window. The SCREEN option inverts the y-axis of the world coordinates, so that (<i>wx1</i> , <i>wy1</i>) represent the upper-left, and (<i>wx2</i> , <i>wy2</i>) the lower-right coordinates of the window.

COMMANDS, STATEMENTS AND FUNCTIONS

Characteristics

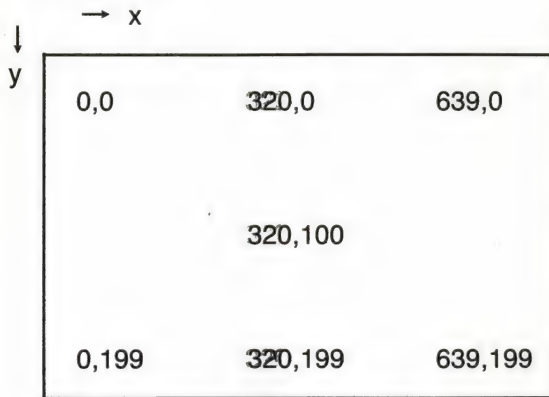
WINDOW allows you to draw lines, graphs, or objects in space not bounded by the physical dimensions of the screen. This is done by using arbitrary programmer-defined coordinates called "world coordinates". The world coordinates are automatically converted to screen coordinates.

If no parameters are given with the WINDOW statement the world coordinates coincide with the screen coordinates.

If you enter:

**NEW
SCREEN 2**

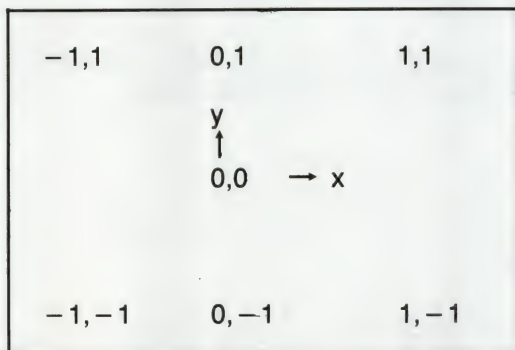
the screen will appear as:



Now enter:

WINDOW (-1,-1)-(1,1)

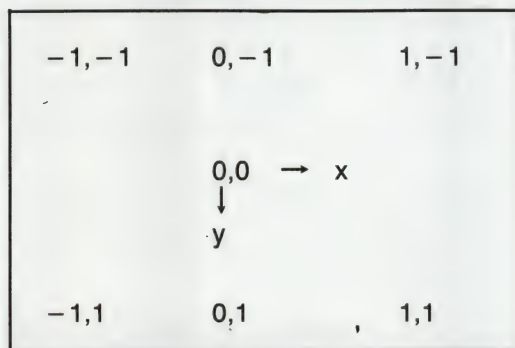
and the screen appears as:



If the variant:

WINDOW SCREEN (-1,-1)-(1,1)

is executed, then the screen appears as:



COMMANDS, STATEMENTS AND FUNCTIONS

Orientation of Axes

If the last executed WINDOW statement specified the SCREEN option, the x- and y-axes of the window have the same orientation as the viewport (the x-axis increasing towards the right and the y-axes towards the bottom); if the statement did not specify the SCREEN option, the y-axis of the window increases towards the top. Therefore, if the same figure is drawn twice using first just WINDOW and then WINDOW SCREEN, the second figure will be a reflection of the first figure in the x-axis.

The following example draws a triangle pointing downwards after the first execution of the statement at line 1000; after the second execution the triangle points upwards.

```
10 SCREEN 1
20 VIEW
30 CLS
40 INPUT X1,Y1,X2,Y2
50 VIEW (X1,Y1)-(X2,Y2),,1
60 WINDOW (-1000,-1000) - (1000,1000)
70 GOSUB 1000
80 WINDOW SCREEN (-1000,-1000) - (1000,1000) : GOSUB 1000
90 END
1000 LINE (-500,500) - (500,500) : LINE - (0,-500):
      LINE - (-500,500) : A$ = INPUT$(1) : CLS : RETURN
```

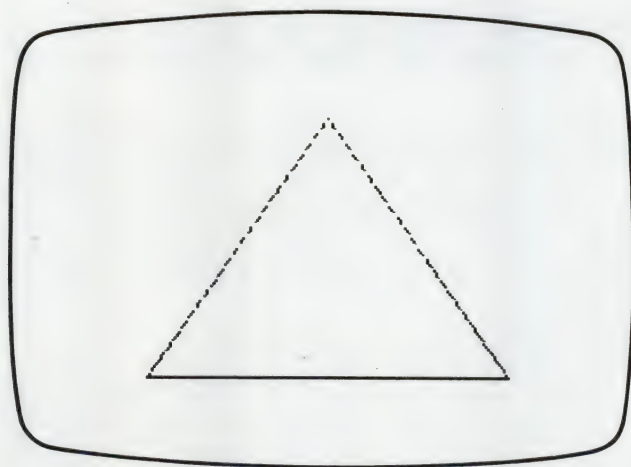
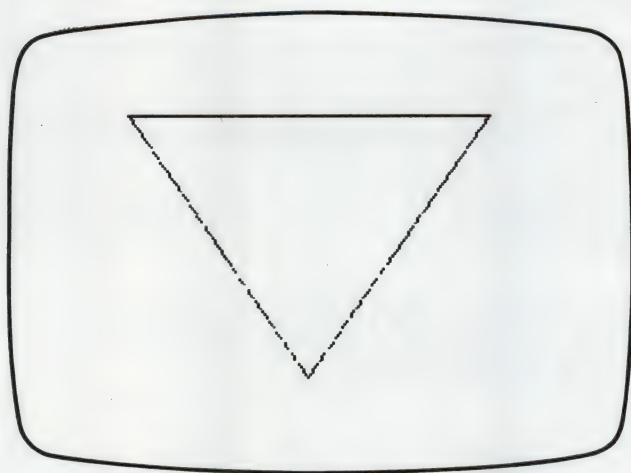


Fig. 8-5 Example of the Orientation of Axes

COMMANDS, STATEMENTS AND FUNCTIONS

Projection of Images onto the Screen

All points drawn with graphics statements in a window (defined using WINDOW) are projected onto the points in the viewport defined using the VIEW statement. If you specify the SCREEN option in the VIEW statement the points drawn in the WINDOW are projected onto the whole screen, but only the points within the current viewport are shown (i.e. clipping).

Example 1

```
10 PI = 3.14159
20 SCREEN 3: CLS: KEY OFF
30 WINDOW (0,-1)-(2*PI, 1)
40 VIEW SCREEN (1,1) - (200,200),,I
50 GOSUB 1000
60 VIEW SCREEN (440,301) - (638,398),,I
70 GOSUB 1000
80 VIEW SCREEN, (1,201) - (638,300),,I
90 GOSUB 1000
100 END
1000 PSET (0,0)
1010 FOR ALPHA = 0 TO 2*PI STEP PI/100
1020 LINE - (ALPHA,SIN(ALPHA))
1030 NEXT
1040 A$ INPUT$(1)
1050 RETURN
```

Draws a sinusoid, and demonstrates clipping, by defining a window (line 30) and viewport using the statement VIEW SCREEN (lines 40, 60 and 80). The statement on line 1040 suspends the program until you press a key.

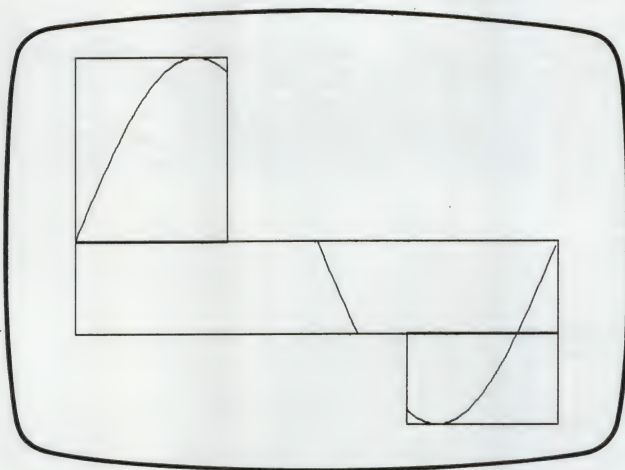


Fig 8-6 Example 1

By varying the areas of the window and the viewport, you change the relationship between the points within them, and you can obtain different screen images from the same design.

If you vary the areas of the window and viewport, while keeping the ratio between the sides constant you will obtain designs that are similar; if you vary the ratio the images will be distorted.

Example 2

```

5 CLS
10 SCREEN 1
20 INPUT X1, Y1, X2, Y2
25 VIEW
30 VIEW (X1,Y1) - (X2,Y2) ,, I
40 WINDOW (-100,-100) - (100,100)
50 GOSUB 1000
60 A$ = INPUT$(1):CLS: GOTO 20
1000 INPUT R
1010 PSET (R,0)

```

COMMANDS, STATEMENTS AND FUNCTIONS

```
1020 PI = 3.14159
1030 FOR ALFA = 0 TO 2*PI STEP PI/100
1040 LINE -(R*COS(ALFA), R*SIN (ALFA))
1050 NEXT : RETURN
```

The above example draws a circle, made up as a polygon with a large number of sides, using a LINE statement in a FOR/NEXT loop. The same circle drawn with a CIRCLE statement would not be distorted if the ratio between the sides of the viewport was varied.

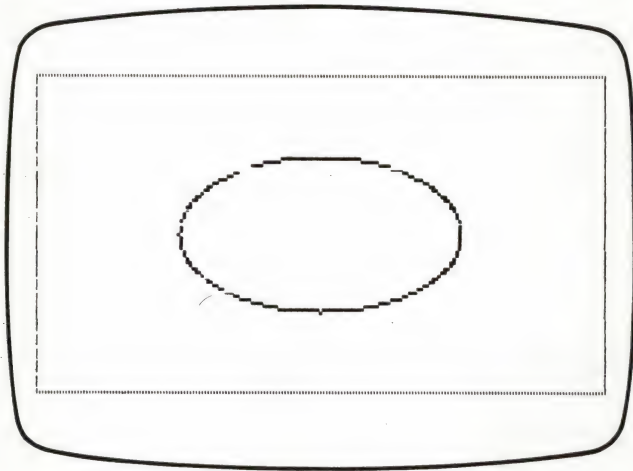


Fig. 8-7 Example 2

If you leave the area of the viewport unchanged and vary the area of the window, you can enlarge or reduce the dimensions of the image displayed or zoom in on a detail.

Example 3

```
10 SCREEN 3
20 VIEW
30 CLS
40 VIEW (1,100) - (638,300),,1
50 WINDOW (-100,-100) - (100,100)
60 GOSUB 1000
70 A$ = INPUT$(1)
80 CLS
90 WINDOW (-40,-40) - (40,40)
```

```

100 GOSUB 1000
110 END
1000 LINE (-50,-30) - (50,30)
1010 CIRCLE (0,0),20
1020 RETURN

```

This example draws a circle crossed by a straight line, and demonstrates the zoom facility, by defining two different windows.

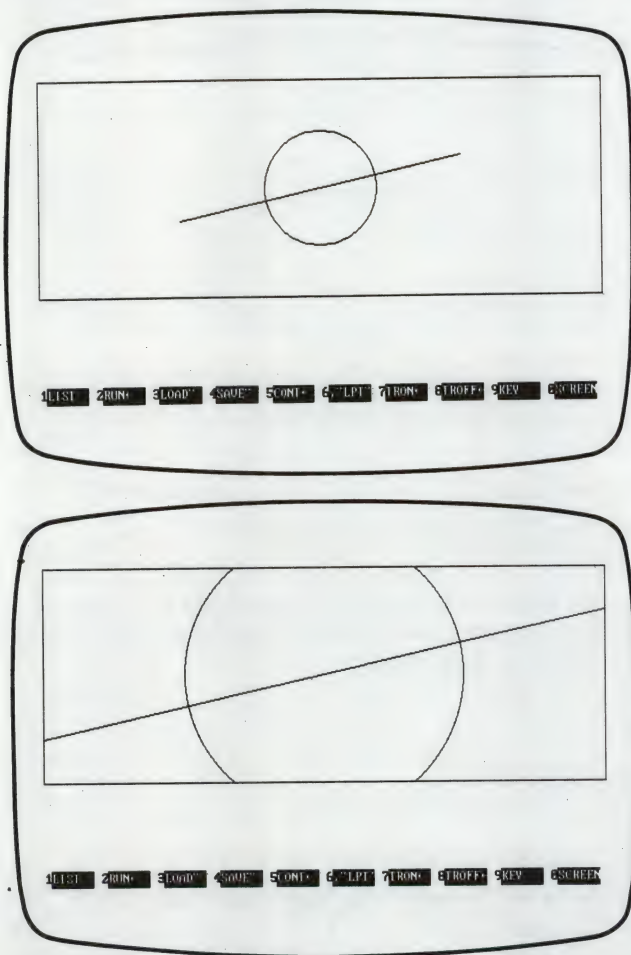


Fig. 8-8 Example 3

COMMANDS, STATEMENTS AND FUNCTIONS

You can also enlarge and reduce images by leaving the window unmodified and varying the area of the viewport.

Example 4

```
10 SCREEN 3 : KEY OFF : CLS
20 PI=3.14159
30 VIEW
40 CLS
50 WINDOW (-100,-100) - (100,100)
60 VIEW (200,1) - (400,200),,I
70 CIRCLE (0,0) , 100
80 PAINT (30,30)
90 VIEW (300,201) - (400,398) ,, I
100 CIRCLE (0,0),100
110 PAINT (30,30)
```

This example draws a circle and paints it, showing how you can define more than one viewport and modify an image on the screen by varying the dimensions of the viewport.

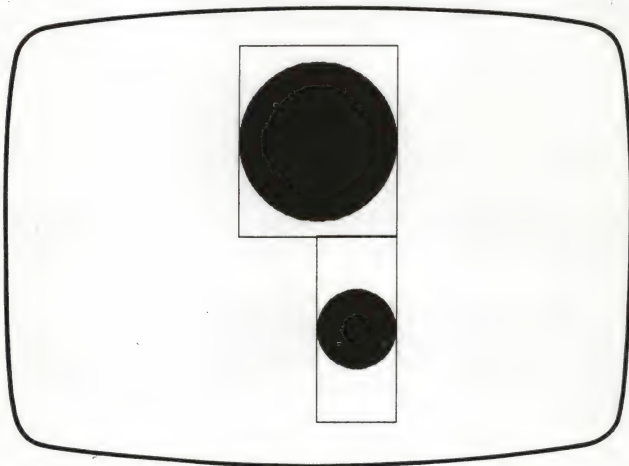


Fig. 8-9 Example 4

Note that no clipping takes place, because the window is not modified and the SCREEN option is not specified in the VIEW statement. The circle is not distorted because the CIRCLE statement is used.

WRITE Statement

Writes data to the screen.

WRITE [*list-of-expressions*]

Where

SYNTAX ELEMENT	MEANING
<i>list-of-expressions</i>	Is a list of numeric and/or string expressions. They must be separated by commas.

Characteristics

The values of the expressions are output to the screen. If no expression is indicated, a blank line is output.

Each item displayed is separated from the last by a comma. Strings are delimited by quotation marks. Numeric values are displayed using the same format as the PRINT statement, but they are not followed by blanks. After the last item in the list is displayed, GW-BASIC inserts a CR LF.

COMMANDS, STATEMENTS AND FUNCTIONS

Example

```
10 A = 80:B = 90:C$ = "THAT'S ALL"  
20 WRITE A,B,C$  
RUN  
80, 90,"THAT'S ALL"  
Ok
```

WRITE # Statement

Writes data to a sequential file.

WRITE # *filenum* , *list-of-expressions*

Where

SYNTAX ELEMENT	MEANING
<i>filenum</i>	Is the number under which the file was OPENed in "O" mode.
<i>list-of-expressions</i>	Is a list of string or numeric expressions. They must be separated by commas.

Characteristics

The difference between the `WRITE #` and `PRINT #` statement is that `WRITE #` inserts commas between the items as they are written to the file and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A `CR LF` sequence is inserted after the last item in the list is written to the file.

Example

```
10 A$ = "CAMERA" : B$ = "93604-1"
```

```
20 WRITE # 1,A$,B$
```

Statement 20 writes the following image to disk:

```
"CAMERA","93604-1"
```

A subsequent `INPUT #` statement, such as

```
30 INPUT # 1,A$,B$
```

would input "CAMERA" to A\$ and "93604-1" to B\$.

A. ERROR CODES AND ERROR MESSAGES

ABOUT THIS APPENDIX

This appendix describes all error codes and messages given by GW-BASIC.

ERROR CODES AND ERROR MESSAGES

NUMBER	MESSAGE
1	<p>NEXT without FOR</p> <p>A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.</p>
2	<p>Syntax error</p> <p>A line is encountered which includes an incorrect sequence of characters (misspelled keyword, unmatched parenthesis, incorrect punctuation, etc). GW-BASIC automatically enters edit mode at the line that caused the error.</p>
3	<p>RETURN without GOSUB</p> <p>A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.</p>
4	<p>Out of DATA</p> <p>A READ statement is executed when there are no DATA statements with unread data remaining in the program.</p>
5	<p>Illegal function call</p> <p>A parameter that is out of range is passed to a numeric or string function. This FC error may also occur as the result of:</p>

NUMBER	MESSAGE
	<ul style="list-style-type: none"> • a negative or unreasonably large subscript • a negative or zero argument with LOG • a negative argument to SQR • a negative mantissa with a noninteger exponent • a call to a USR function for which the starting address has not yet been given • an improper argument to MID\$, LEFT\$, RIGHT\$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING\$, SPACE\$, INSTR, or ON...GOTO • a negative record number used with GET(Files) or PUT(Files) statements
6	<p>Overflow</p> <p>The result of a calculation is too large to be represented in GW-BASIC number format. If underflow occurs, the result is zero and execution continues without an error.</p>
7	<p>Out of memory</p> <p>A program is too big, or has too many loops, subroutines, variables, or has expressions that are too complicated to evaluate.</p>
8	<p>Undefined line number</p> <p>A nonexistent line is referenced in a GOTO, GOSUB, IF...THEN...ELSE, or DELETE statement.</p>

ERROR CODES AND ERROR MESSAGES

NUMBER	MESSAGE
9	<p>Subscript out of range</p> <p>An array element is referenced either with a subscript that is outside the dimensions of the array or with the wrong number of subscripts.</p>
10	<p>Duplicate Definition</p> <p>Two DIM statements are given for the same array; or a DIM statement is given for an array after the default dimension of 10 has been established for that array; or an OPTION BASE is given after an array has been dimensioned.</p>
11	<p>Division by zero</p> <p>A division by zero is encountered in an expression; or, the value zero has been raised to a negative power. In the former case, the result is machine infinity (with the appropriate sign); in the latter case, the result is positive machine infinity. In both cases execution continues.</p>
12	<p>Illegal direct</p> <p>A statement that is illegal in direct mode is entered as a direct mode command.</p>
13	<p>Type mismatch</p> <p>A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.</p>

NUMBER	MESSAGE
14	<p>Out of string space</p> <p>String variables have caused GW-BASIC to exceed the amount of free memory remaining. GW-BASIC will allocate string space dynamically, until it runs out of memory.</p>
15	<p>String too long</p> <p>An attempt is made to create a string more than 255 characters long.</p>
16	<p>String formula too complex</p> <p>A string expression is too long or too complex to be processed. It should be broken into smaller expressions.</p>
17	<p>Can't continue</p> <p>An attempt is made to continue a program that:</p> <ul style="list-style-type: none"> • has halted due to an error • has been modified during a break in execution • does not exist
18	<p>Undefined user function</p> <p>AUSR function is called before the function definition (DEF statement) is given.</p>

ERROR CODES AND ERROR MESSAGES

NUMBER	MESSAGE
19	No RESUME An error handling routine is entered but contains no RESUME statement.
20	RESUME without error A RESUME statement is encountered before an error handling routine is entered.
22	Missing operand An expression contains an operator with no operand following it.
23	Line buffer overflow An attempt has been made to input a line that has too many characters.
24	Device Timeout GW-BASIC did not receive information from an I/O device within a predetermined amount of time.
25	Device fault An incorrect device designation has been entered.
26	FOR without NEXT A FOR statement was encountered without a matching NEXT statement.

NUMBER	MESSAGE
27	<p>Out of Paper</p> <p>The printer is out of paper or is not switched on. Insert paper, ensure power is switched on and continue.</p>
29	<p>WHILE without WEND</p> <p>A WHILE statement does not have a matching WEND statement.</p>
30	<p>WEND without WHILE</p> <p>A WEND statement was encountered without a matching WHILE statement.</p>
50	<p>FIELD overflow</p> <p>A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.</p>
51	<p>Internal error</p> <p>An internal malfunction has occurred in GW-BASIC.</p>
52	<p>Bad file number</p> <p>A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.</p>

ERROR CODES AND ERROR MESSAGES

NUMBER	MESSAGE
53	<p>File not found</p> <p>A LOAD, KILL, NAME or OPEN statement/command references a file that does not exist on the current disk.</p>
54	<p>Bad file mode</p> <p>An attempt is made to use PUT(Files), GET(Files), to LOAD a random file, or to execute an OPEN statement with a file mode other than I, O, or R.</p>
55	<p>File already open</p> <p>A sequential output mode OPEN statement is issued for a file that is already open; or a KILL command is given for a file that is open.</p>
57	<p>Device I/O Error</p> <p>An I/O error occurred on a disk I/O operation. It is a fatal error; i.e., the operating system cannot recover from the error.</p>
58	<p>File already exists</p> <p>The filename specified in a NAME command is identical to a filename already in use on the disk.</p>
61	<p>Disk full</p> <p>All disk storage space is in use.</p>

NUMBER	MESSAGE
62	<p>Input past end</p> <p>An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end-of-file.</p>
63	<p>Bad record number</p> <p>In a PUT(Files) or GET(Files) statement, the record number is either greater than the maximum allowed (16,777,215) or equal to zero.</p>
64	<p>Bad file name</p> <p>An illegal form is used for the filename with a LOAD, SAVE, KILL, or OPEN statement/command (e.g., a filename with too many characters).</p>
66	<p>Direct statement in file</p> <p>A direct statement is encountered while LOADING an ASCII-format file. The LOAD is terminated.</p>
67	<p>Too many files</p> <p>An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.</p>
68	<p>Device Unavailable</p> <p>An attempt was made to open a file to a non-existent device. It may be that hardware did not exist to support the device, such as LPT2: or LPT3:.</p>

ERROR CODES AND ERROR MESSAGES

NUMBER	MESSAGE
	or was disabled by the user. This occurs if an OPEN "COM1:... statement is executed after the user has disabled RS232 support via the /C:0 switch directive in the GWBASIC command.
69	<p>Communication buffer overflow</p> <p>Occurs when a communication input statement is executed but the input queue was already full. Use an ON ERROR GOTO statement to retry the input when this condition occurs. Subsequent inputs will attempt to clear this fault, unless characters continue to be received faster than the program can process them. In this case several options are available:</p> <ul style="list-style-type: none">• increase the size of the COM receive buffer via the /C: switch• implement a "hand-shaking" protocol with the host/satellite such as XON/XOFF to turn transmit off long enough to catch up• use a lower baud rate for transmit and receive
70	<p>Permission Denied</p> <p>This occurs when an attempt is made to write to a diskette that is write protected. Use an ON ERROR GOTO statement to detect this situation and request user action.</p> <p>Errors 71, 72, and 74 are other possible "hard" disk errors.</p> <p>The error also occurs during LOCK, UNLOCK and OPEN statements, if access to a specified file is restricted or if MS-NET is not installed.</p>

NUMBER	MESSAGE
71	<p>Disk not ready</p> <p>Occurs when the diskette drive door is open, or a diskette is not in the drive. Again use an ON ERROR GOTO statement to recover.</p>
72	<p>Disk media error</p> <p>Occurs when a hardware or media fault is detected. This usually indicates damaged media. Copy any existing files to a new diskette and reformat the damaged diskette. FORMAT will flag the bad tracks and place them in a file "badtrack". The remainder of the diskette is now usable.</p>
74	<p>Rename across disks</p> <p>An attempt was made to rename a file with a new drive designation.</p>
75	<p>Path/File Access Error</p> <p>During an OPEN, MKDIR, CHDIR, or RMDIR operation, MS-DOS was unable to make a correct Path to Filename connection. The operation is not completed.</p>
76	<p>Path not found</p> <p>During an OPEN, MKDIR, CHDIR, or RMDIR operation, MS-DOS was unable to find the path specified. The operation is not completed.</p>

ERROR CODES AND ERROR MESSAGES

NUMBER	MESSAGE
**	<p>Can't continue after SHELL</p> <p>No error number. Upon returning from a Child process, the SHELL statement discovers that there is not enough memory for GW-BASIC to continue. GW-BASIC closes any open files and exits to MS-DOS.</p>

B. MATHEMATICAL FUNCTIONS

ABOUT THIS APPENDIX

This appendix describes how to calculate commonly used mathematical functions that are not intrinsic to GW-BASIC.

MATHEMATICAL FUNCTIONS

The following shows how to express commonly used mathematical functions using those functions intrinsic to GW-BASIC.

FUNCTION	GW-BASIC EQUIVALENT
SECANT	$\text{SEC}(x) = 1/\text{COS}(x)$ when $x < > 1.570796$
COSECANT	$\text{CSC}(x) = 1/\text{SIN}(x)$ when $x < > 0$
COTANGENT	$\text{COT}(x) = 1/\text{TAN}(x)$ when $x < > 0$
INVERSE SINE	$\text{ARCSIN}(x) = \text{ATN}(x/\text{SQR}(1-x^2))$
INVERSE COSINE	$\text{ARCCOS}(x) = 1.570796 - \text{ATN}(x/\text{SQR}(1-x^2))$ when $\text{ABS}(x) < 1$
INVERSE SECANT	$\text{ARCSEC}(x) = \text{ATN}(\text{SQR}(x^2-1))$ + $\text{SGN}(\text{SGN}(x)-1) * 1.570796$ when $\text{ABS}(x) > 1$
INVERSE COSECANT	$\text{ARCCSC}(x) = \text{ATN}(1/\text{SQR}(x^2-1))$ + $(\text{SGN}(x)-1) * 1.570796$ when $\text{ABS}(x) > 1$
INVERSE COTANGENT	$\text{ARCCOT}(x) = 1.570796 - \text{ATN}(x)$
HYPERBOLIC SINE	$\text{SINH}(x) = (\text{EXP}(x) - \text{EXP}(-x))/2$
HYPERBOLIC COSINE	$\text{COSH}(x) = (\text{EXP}(x) + \text{EXP}(-x))/2$
HYPERBOLIC TANGENT	$\text{TANH}(x) = (\text{EXP}(x) - \text{EXP}(-x)) / (\text{EXP}(x) + \text{EXP}(-x))$
HYPERBOLIC SECANT	$\text{SECH}(x) = 2/(\text{EXP}(x) + \text{EXP}(-x))$
HYPERBOLIC COSECANT	$\text{CSCH}(x) = 2/(\text{EXP}(x) - \text{EXP}(-x))$ when $x < > 0$

FUNCTION	GW-BASIC EQUIVALENT
HYPERBOLIC COTANGENT	$\text{COTH}(x) = (\text{EXP}(x) + \text{EXP}(-x)) / (\text{EXP}(x) - \text{EXP}(-x))$ when $x < > 0$
INVERSE HYPERBOLIC SINE	$\text{ARCSINH}(x) = \text{LOG}(x + \text{SQR}(x^2 + 1))$
INVERSE HYPERBOLIC COSINE	$\text{ARCCOSH}(x) = \text{LOG}(x + \text{SQR}(x^2 - 1))$ when $x \geq 1$
INVERSE HYPERBOLIC TANGENT	$\text{ARCTANH}(x) = \text{LOG}((1 + x)/(1 - x))/2$ when $\text{ABS}(x) < 1$
INVERSE HYPERBOLIC SECANT	$\text{ARCSECH}(x) = \text{LOG}((\text{SQR}(1 - x^2) + 1)/x)$ when $0 < x \leq 1$
INVERSE HYPERBOLIC COSECANT	$\text{ARCCSCH}(x) = \text{LOG}((\text{SGN}(x) * \text{SQR}(x^2 + 1) + 1)/x)$ when $x > 0$
INVERSE HYPERBOLIC COTANGENT	$\text{ARCCOTH}(x) = \text{LOG}((x + 1)/(x - 1))/2$ when $\text{ABS}(x) > 1$
LOGARITHM TO BASE 'a'	$\text{LOGA}(x) = \text{LOG}(x) / \text{LOG}(a)$ when $a > 0$ and $x > 0$

You can define a derived mathematical function in your program by use of a DEF FN statement, to avoid coding the formula each time you need it.

Note that both 'x' and 'a' can be any numeric constant, variable, array element, function or expression. Any values of 'x' or 'a' that would cause error messages are specified.

C. ASCII CHARACTER CODES

ABOUT THIS APPENDIX

This appendix provides a table of ASCII characters.

ASCII CHARACTER CODE

This table shows the decimal and hexadecimal codes and the symbols displayed for the 256 elements of the standard ASCII character set.

DEC	HEX	CHARACTER	DEC	HEX	CHARACTER	DEC	HEX	CHARACTER	DEC	HEX	CHARACTER
000	00	BLANK (NULL)	016	10	► (DLE)	032	20	BLANK (SPACE)	048	30	0
001	01	😊 (SOH)	017	11	◄ (DC1)	033	21	!	049	31	1
002	02	😬 (STX)	018	12	↕ (DC2)	034	22	”	050	32	2
003	03	♥ (ETX)	019	13	!! (DC3)	035	23	#	051	33	3
004	04	♦ (EOT)	020	14	⏏ (DC4)	036	24	\$	052	34	4
005	05	♣ (ENQ)	021	15	§ (NAC)	037	25	%	053	35	5
006	06	♠ (ACK)	022	16	▬ (SYN)	038	26	&	054	36	6
007	07	• (BEL)	023	17	↕ (ETB)	039	27	'	055	37	7
008	08	■ (BS)	024	18	↑ (CAN)	040	28	(056	38	8
009	09	○ (HT)	025	19	↓ (EM)	041	29)	057	39	9
010	0A	◉ (LF)	026	1A	→ (SUB)	042	2A	*	058	3A	:
011	0B	♂ (VT)	027	1B	← (ESC)	043	2B	+	059	3B	;
012	0C	♀ (FF)	028	1C	└ (FS)	044	2C	,	060	3C	<
013	0D	🎵 (CR)	029	1D	↔ (GS)	045	2D	—	061	3D	=
014	0E	🎵 (SO)	030	1E	▲ (RS)	046	2E	.	062	3E	>
015	0F	☀ (SI)	031	1F	▼ (US)	047	2F	/	063	3F	?

Tab. C-1 Standard ASCII Character Set

DEC	HEX	CHARACTER	DEC	HEX	CHARACTER	DEC	HEX	CHARACTER	DEC	HEX	CHARACTER
064	40	@	080	50	P	096	60	'	112	70	p
065	41	A	081	51	Q	097	61	a	113	71	q
066	42	B	082	52	R	098	62	b	114	72	r
067	43	C	083	53	S	099	63	c	115	73	s
068	44	D	084	54	T	100	64	d	116	74	t
069	45	E	085	55	U	101	65	e	117	75	u
070	46	F	086	56	V	102	66	f	118	76	v
071	47	G	087	57	W	103	67	g	119	77	w
072	48	H	088	58	X	104	68	h	120	78	x
073	49	I	089	59	Y	105	69	i	121	79	y
074	4A	J	090	5A	Z	106	6A	j	122	7A	z
075	4B	K	091	5B	[107	6B	k	123	7B	{
076	4C	L	092	5C	\	108	6C	l	124	7C	
077	4D	M	093	5D]	109	6D	m	125	7D	}
078	4E	N	094	5E	^	110	6E	n	126	7E	~
079	4F	O	095	5F	_	111	6F	o	127	7F	Δ

Tab. C-1 Standard ASCII Character Set (cont.)

ASCII CHARACTER CODE

DEC	HEX	CHARACTER	DEC	HEX	CHARACTER	DEC	HEX	CHARACTER	DEC	HEX	CHARACTER
128	80	Ç	144	90	É	160	A0	á	176	B0	▒
129	81	ü	145	91	æ	161	A1	í	177	B1	▒
130	82	é	146	92	Æ	162	A2	ó	178	B2	▒
131	83	â	147	93	ô	163	A3	ú	179	B3	
132	84	ä	148	94	Ö	164	A4	ñ	180	B4	┐
133	85	à	149	95	ò	165	A5	Ñ	181	B5	≡
134	86	á	150	96	û	166	A6	ä	182	B6	≡
135	87	ç	151	97	ù	167	A7	ö	183	B7	⌋
136	88	ê	152	98	ÿ	168	A8	ı	184	B8	⌋
137	89	ë	153	99	Ö	169	A9	┐	185	B9	≡
138	8A	è	154	9A	Ü	170	AA	┐	186	BA	
139	8B	ï	155	9B	¢	171	AB	1/2	187	BB	⌋
140	8C	î	156	9C	£	172	AC	1/4	188	BC	⌋
141	8D	ì	157	9D	¥	173	AD	ı	189	BD	⌋
142	8E	Ä	158	9E	Pt	174	AE	«	190	BE	≡
143	8F	Å	159	9F	f	175	AF	»	191	BF	┐

Tab. C-1 Standard ASCII Character Set. (cont)

DEC	HEX	CHARACTER	DEC	HEX	CHARACTER	DEC	HEX	CHARACTER	DEC	HEX	CHARACTER
192	C0	┐	208	D0	┘	224	E0	α	240	F0	≡
193	C1	└	209	D1	┐	225	E1	β	241	F1	±
194	C2	┘	210	D2	└	226	E2	Γ	242	F2	≥
195	C3	┌	211	D3	┘	227	E3	Π	243	F3	≤
196	C4	—	212	D4	┐	228	E4	Σ	244	F4	∫
197	C5	+	213	D5	┘	229	E5	σ	245	F5	ℑ
198	C6	≡	214	D6	└	230	E6	μ	246	F6	÷
199	C7	┌	215	D7	┘	231	E7	τ	247	F7	≈
200	C8	┘	216	D8	≠	232	E8	φ	248	F8	°
201	C9	┘	217	D9	└	233	E9	⊖	249	F9	•
202	CA	┘	218	DA	└	234	EA	Ω	250	FA	•
203	CB	┘	219	DB	■	235	EB	δ	251	FB	√
204	CC	┘	220	DC	■	236	EC	∞	252	FC	ℓ
205	CD	=	221	DD	■	237	ED	∅	253	FD	₂
206	CE	≡	222	DE	■	238	EE	∈	254	FE	■
207	CF	┘	223	DF	■	239	EF	∩	255	FF	BLANK 'FF'

Tab. C-1 Standard ASCII Character Set (cont.)

ASCII CHARACTER CODE

For Denmark and Norway certain characters differ from the standard. These characters and their decimal and hexadecimal codes are shown in the following table.

DEC	HEX	CHARACTER
155	9B	ø
157	9D	.Ø
158	9E	Ł
159	9F	İ
166	A6	õ
167	A7	Õ
169	A9	ã
170	AA	Ã
171	AB	ℓ
172	AC	'n
174	AE	3
175	AF	Ø

Tab. C-2 Characters for Denmark and Norway.

D. CONVERSION TABLES

ABOUT THIS APPENDIX

This appendix shows the conversion tables for hexadecimal to decimal and for decimal to binary, octal and hexadecimal.

CONVERSION TABLES

HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4096	1	256	1	16	1	1
2	8192	2	512	2	32	2	2
3	12288	3	768	3	48	3	3
4	16384	4	1024	4	64	4	4
5	20480	5	1280	5	80	5	5
6	24576	6	1536	6	96	6	6
7	28672	7	1792	7	112	7	7
8	32768	8	2048	8	128	8	8
9	36864	9	2304	9	144	9	9
A	40960	A	2560	A	160	A	10
B	45056	B	2816	B	176	B	11
C	49152	C	3072	C	192	C	12
D	53248	D	3328	D	208	D	13
E	57344	E	3584	E	224	E	14
F	61440	F	3840	F	240	F	15
4		3		2		1	
BYTE				BYTE			

Tab. D-1 Conversion Table for Hexadecimal to Decimal

DECIMAL	BINARY	OCTAL	HEX
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Tab. D-2 Conversion Table for Decimal, Binary, Octal and Hexadecimal

E. CONVERSION OF PROGRAMS TO GW-BASIC

ABOUT THIS APPENDIX

This appendix outlines the criteria to be followed in order to convert programs written in other BASICs to GW-BASIC.

CONTENTS

INTRODUCTION	E-1
STRING DIMENSIONING	E-1
LENGTH OF STRINGS	E-1
SUBSTRINGS	E-1
CONCATENATION	E-2
MAT FUNCTIONS	E-2
MULTIPLE ASSIGNMENTS	E-2
MULTIPLE STATEMENTS	E-3
PEEKs AND POKEs	E-3
IF...THEN[...ELSE]	E-3
FILE I/O	E-3
GRAPHICS	E-4

CONVERSION OF PROGRAMS TO GW-BASIC

INTRODUCTION

The GW-BASIC language bears a similarity to many BASICs. Your personal computer will support programs written for an extensive variety of microcomputers. For programs written in a BASIC other than GW-BASIC, some minor adjustments may be necessary before running them. This appendix highlights some specific areas to examine when converting programs.

STRING DIMENSIONING

LENGTH OF STRINGS

GW-BASIC strings are of variable lengths. Therefore, all statements that declare the length of strings should be deleted. For example, in a statement which dimensions a string array for 'J' elements of lengths 'I' such as:

```
DIM A$(I,J)
```

the conversion for GW-BASIC would be:

```
DIM A$(J)
```

SUBSTRINGS

In GW-BASIC the following functions are used to take substrings of strings:

```
LEFT$  
MID$  
RIGHT$
```

Other forms, such as:

A\$(I) (to access the Ith character in A\$) or,
A\$(I,J) (to take a substring of A\$ from position I to J) should be changed as follows:

Other BASICs

X\$ = A\$(I)
X\$ = A\$(I,J)

GW-BASIC

X\$ = MID\$(A\$,I,1)
X\$ = MID\$(A\$,I,J-I + 1)

If the substring reference is on the left side of an assignment and X\$ is used to replace characters in A\$, then the conversion should be carried out as follows:

Other BASICs

A\$(I) = X\$
A\$(I,J) = X\$

GW-BASIC

MID\$(A\$,I,1) = X\$
MID\$(A\$,I,J-I + 1) = X\$

CONCATENATION

GW-BASIC uses a plus (+) sign to denote string concatenation. Other BASICs use a comma (,) or an ampersand (&) which should be altered accordingly.

MAT FUNCTIONS

Some BASICs incorporate MAT functions for array handling. To convert a program which uses these functions to the GW-BASIC environment, it is necessary to rewrite the program including FOR...NEXT loops.

MULTIPLE ASSIGNMENTS

Some BASICs allow the following syntax:

```
10 LET D = E = 0
```

to set D and E equal to zero. GW-BASIC interprets the second equal sign as a logical operator and sets D equal to -1 if E was equal to 0. This statement should therefore be broken up into two assignment statements as follows:

```
10 D = 0:E = 0
```

CONVERSION OF PROGRAMS TO GW-BASIC

MULTIPLE STATEMENTS

Multiple statements on a GW-BASIC line must always be separated by colons (:), unlike some other BASICs which use a backslash (\) instead.

PEEKs AND POKEs

The execution of programs containing PEEK and POKE instructions may vary from machine to machine. It is therefore necessary to analyse the purpose of these instructions in other BASIC programs before translating the same functions into GW-BASIC.

IF...THEN...[ELSE...]

Not all BASICs feature the optional ELSE clause which is performed in the event of a test proving false.

For example, a BASIC statement may originally be:

```
10 IF D = E THEN 30
20 PRINT "NOT EQUAL" : GOTO 40
30 PRINT "EQUAL"
40 REM CONTINUE
```

The above statement sequence will work correctly, but it may be optimized in GW-BASIC as follows:

```
10 IF D = E THEN PRINT "EQUAL" ELSE PRINT "NOT EQUAL"
20 REM CONTINUE
```

FILE I/O

In some BASICs, the I/O to disk may differ from GW-BASIC. Refer to Chapter 4 for more information.

GRAPHICS

Selecting screen attributes and drawing objects on the screen can vary from BASIC to BASIC. Refer to Chapter 5 for more information on graphics.

F. MEMORY MANAGEMENT

ABOUT THIS APPENDIX

This appendix contains specific technical information pertaining to GW-BASIC.

It includes:

- How GW-BASIC allocates variables
- Internal representation of floating point numbers
- Memory map
- The File Control Block Structure

CONTENTS

ALLOCATION OF VARIABLES	F-1
INTERNAL REPRESENTATION OF FLOATING POINT NUMBERS	F-3
FILE CONTROL BLOCK	F-4
MEMORY MAP	F-8

MEMORY MANAGEMENT

ALLOCATION OF VARIABLES

Each variable defined is associated with a record allocated in the GW-BASIC Data Segment. The record is made up of a field structure, each field contains an attribute of the variable (type, name, string descriptor) or its value. A record associated with a numeric variable contains its value, while a record associated with a string variable contains a field (the string descriptor) which points to the address, in the String Space, in which the value of the variable is stored.

The following table shows the way in which a numeric or string variable is allocated in memory. In particular, the first column gives the offset of the field with respect to the start address of the record, the second column gives the length (in bytes), the third column the contents, and the fourth provides other information.

OFFSET	LENGTH	DESCRIPTION	COMMENTS
0	1	Type	The value of the variable type is: 2 (integers) 3 (strings) 4 (single-precision numbers) 5 (double-precision numbers)

Tab. F-1 Allocation of Variables

OFFSET	LENGTH	DESCRIPTION	COMMENTS
1	3 to $n + 3$	Name	<ul style="list-style-type: none"> - Bytes 1 and 2 contain the first 2 characters of the name - Byte 3 contains a number indicating how many more characters are in the name - Bytes 4 to n contain the additional characters of the name. Three bytes are reserved for the name of the variable, even if the name is only one or two characters long. In this case the data (or string descriptor) begins with an offset of 4.
$4 + n$	2	Integer Value	Integers are stored low byte first, then high byte
	3	String Descriptor	<ul style="list-style-type: none"> - First byte contains the string length (0-255) - Second byte is the low byte of the offset to the start address of GW-BASIC's data segment - Third byte is the high byte of the offset to the start address of GW-BASIC's data segment
	4	Single Precision Value	In floating point format
	8	Double Precision Value	In floating point format

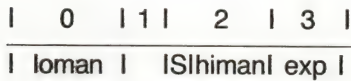
Tab. F-1 Allocation of Variables (cont.)

MEMORY MANAGEMENT

INTERNAL REPRESENTATION OF FLOATING POINT NUMBERS

The following section describes the internal representation of numbers in GW-BASIC.

Single Precision - 24 bit mantissa



where:

loman = the low mantissa

S = the sign

himan = the high mantissa

exp = the exponent

man = himan:.....:loman

If exp = 0, then number = 0

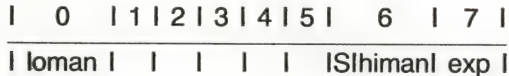
If exp < > 0, then the mantissa is normalized and
number = sign * .1 man * 2 ** (exp - 80h)

That is, in single precision (hex notation - bytes low to high)

00000080 = .5

00008080 = -.5

Double Precision - 56 bit mantissa



FILE CONTROL BLOCK

The following table shows the information held in the File Control Block (FCB). The offsets are with respect to the address returned by the VARPTR function. GW-BASIC's File Control Block is different from that of MS-DOS.

OFFSET	LENGTH	NAME	MEANING
0	1	Mode	The mode in which file was opened: 1 - Input 2 - Output 4 - Random Access 16 - Append 32 - Reserved 128 - Reserved
1	38	FCB	MS-DOS File Control Block.
39	2	CURLOC	Number of sectors written to or read (for sequential files); number of last record written to or read + 1 (for random access files).
41	1	ORNOFS	Number of bytes per sector

Tab. F-2 File Control Block

MEMORY MANAGEMENT

OFFSET	LENGTH	NAME	MEANING
42	1	NMLOFS	Number of bytes left in INPUT buffer
43	3	-	Reserved
46	1	DEVICE	Number of the device: 0-9 Disk Drives (A: - J:) 255 Keyboard 254 SCRNL: 253 LPT1: 252 CAS1: 251 COM1: 250 COM2: 249 LPT2: 248 LPT3:
47	1	WIDTH	Line width of the device
48	1	POS	Buffer position for PRINT #
49	1	FLAGS	Internal use during LOAD/SAVE. Not used for file data
50	1	OUTPOS	Output position used in tabulation operations

Tab. F-2 File Control Block (cont.)

OFFSET	LENGTH	NAME	MEANING
51	128	BUFFER	Data Buffers, used for the transfer of data between MS-DOS and GW-BASIC. This offset must be used for examining data in sequential I/O.
179	2	VRECL	Record length for random access files, set in the OPEN statement. Default value: 128.
181	2	PHYREC	Number of current physical record
183	2	LOGREC	Number of current logical record
185	1	-	Reserved
186	2	OUTPOS	Position used by PRINT#, INPUT# and WRITE#. Only used for disk files.

Tab. F-2 File Control Block (cont.)

MEMORY MANAGEMENT

OFFSET	LENGTH	NAME	MEANING
188	n	FIELD	Current Buffer Field. Its length, in bytes, is set by the /S: switch in the GWBASIC Command. VRECL bytes are transferred between BUFFER and FIELD in I/O operations. This offset must be used to examine the data of a I/O operation for random access files.

Tab F-2 File Control Block (cont.)

MEMORY MAP

The map below illustrates the system memory structure.

MS-DOS Segment	System interrupt vectors, variables, etc
	MS-DOS (approximately 12 K)
	MS-DOS workarea
GW-BASIC Code Segment	GW-BASIC code (approximately 60 K)
COM Buffer Segment	GW-BASIC Communications buffer(s)
GW-BASIC Data Segment	Interpreter Workarea (approximately 3 K)
	GW-BASIC Program
	Scalar Variables
	Arrays
	String Space
	GW-BASIC stack 128 Bytes (or set by the CLEAR command)
...	
System Segment	System memory, screen memory
	ROM

Fig. F-3 Memory Map

MEMORY MANAGEMENT

In the GW-BASIC workarea there are some variables available to the user, which may be accessed via PEEK or POKE. These variables are not at the actual addresses specified: PEEK and POKE map the specified addresses to the actual locations.

The start address of the GW-BASIC Data Segment may be obtained by examining the word at absolute location 0:510H. (This is an actual address).

This information is not required to execute PEEKs and POKEs; a DEF SEG statement sets up the right segment. In this case, the following locations are of significance:

ADDRESS (Decimal)	OFFSET (Hex)	MEANING
46	2EH	Current line number
839	347H	Current Error line number
48	30H	Offset of start of Program
856	358H	Offset of start of Variables
106	6AH	Keyboard buffer PEEK/POKE location

bit map: An area in memory, the bits of which are associated with points on the screen. If only one bit is associated with one point on the screen, then you can display two colors (black or white); if two bits are associated to one point on the screen you can display four colors.

boolean value: A numeric value that GW-BASIC uses (in a statement or a command) as a "true" or "false" condition to direct program flow or for other purposes.

boot: The loading of the operating system into memory.

built-in function: See intrinsic function.

call: The branching or transfer of control to a specified subroutine.

carriage return character (CR): A character that moves the cursor or the print head position to the beginning of the next line. Entering CR when you finish typing a GW-BASIC line, passes the line to GW-BASIC for processing.

clipping: The graphics statements use line clipping, i.e. lines that cross the screen or viewport are "clipped" at the edges of the viewing area.

color code: A number in the range 0 to 15 that identifies a color.

color number: In graphics, a number, in the range 0 to 3, that selects the actual color from a palette.

command level: GW-BASIC is at command level when Ok appears on the screen, i.e. when it is waiting for the user to enter an immediate or program line.

comment: A remark inserted in a program for documentation purposes. In GW-BASIC, a comment may be entered by REM or a single quote (') followed by the comment string. The single quote (') also allows the insertion of comments at the end of a GW-BASIC line.

concatenation: The operation that joins a string to another string by the use of the operator "+".

constant: A value which does not change during program execution. A constant may be a string or a numeric constant. In the latter case it may be an integer, a single-precision or a double-precision number.

GLOSSARY

coordinates: Numbers that are used to specify a location on the screen. They may be text coordinates to identify a character or the cursor (expressed in terms of rows and columns) or graphics coordinates to identify a pixel (expressed as x and y Cartesian coordinates)

current directory: The directory you are working on. You may change the current directory by the CHDIR command. Just after formatting a disk the Root directory is the current directory.

current line: The line you are working on, or the line you have just entered, or the line where an error has occurred.

current point: See the "last referenced point".

current program: The program currently in memory.

current segment: The memory segment defined by the last DEF SEG statement that has been executed, or the GW-BASIC Data Segment.

current viewport: The viewport you are working on. To change viewport you must use a VIEW statement.

cursor: A movable marker which indicates a screen position. There are three types of cursor: the overwrite, the insert and the user cursor. The shape and blinkrate of the overwrite and user cursors are programmable. The user cursor is not visible but can be made visible using the LOCATE Statement.

debug: To locate and correct errors in a program.

default: The value of a parameter that will be assumed by the system, if no value is entered by the user.

default drive: When a file specifier or a pathname is given, the default drive is the one that was the default in MS-DOS before GW-BASIC was invoked. At initialization the default drive is the drive from which MS-DOS is loaded. The default drive may only be changed in MS-DOS entering the driver specifier (A:, B:, C:, etc ...) either before the name of an MS-DOS command, or alone.

destination variable: The variable to the left of the equal sign in an assignment statement.

device: A peripheral. Examples of devices are: the printer, the keyboard, the screen, an I/O port, etc.

direct access: File access method by which each record is directly addressable. The ability to read or write information at any location within a storage device.

direct line: See "immediate line".

direct mode: See "immediate mode".

directory: The directory contains the names of files on the disk, along with information that tells MS-DOS where to find the file.

disk: Is a generic term to specify either a hard-disk or a diskette.

diskette: A 5 1/4 inch mini floppy disk.

double precision: This is the maximum precision GW-BASIC can handle. If a number contains more than 7 digits it is a double-precision number.

drive: Synonymous with disk drive. May be specified by A: (first diskette drive), B: (second diskette drive), C: (hard-disk drive), etc..

dummy argument: A fictitious parameter in a function or statement or command. A value must be entered, but it is ignored by GW-BASIC.

edit: To modify a GW-BASIC line displayed on the screen.

end of file (EOF): A "marker" following the last record in a file.

error trapping: When an error occurs, the control of the program may be automatically directed to a specified error handling routine, rather than to the standard routine. A user error handling routine should check for all the particular errors the user wishes to recover from, and indicate what to do in each case.

event trapping: When a certain event occurs, the control of the program may be automatically directed to a specified trapping routine. The trap routine, after servicing the event, executes a RETURN statement that causes the program to resume execution. The events that can be trapped are receipt of characters from an RS-232-C port, detection of certain keystrokes, time passage, and emptying of the background music queue.

GLOSSARY

expression: An algorithm returning a single numeric value (numeric, relational or logical expressions) or a concatenation of strings returning a string value (string expression).

extended code: The extended ASCII code is returned by the INKEY\$ function if a key (or key combination) is entered, that cannot be associated with a standard ASCII code (See the "MS-DOS User Guide").

field: The area in a record, used to allocate a particular data item.

file: A collection of records. The records of a file may be accessed by GW-BASIC sequentially (one after the other) or randomly (by record number).

filename: The name assigned to a file. It may include an extension of three characters separated from the filename with a period. (BAS, BAT, COM, or EXE extensions have special meanings.)

File number: A number, in the range 1 to 255, associated with a file when it is opened. Used by all I/O statements to refer to a file.

file specifier: Unique file identifier. It includes the filename, possibly preceded by a device specifier (A:, B:, C:, ...). If no device is specified the current drive is assumed.

fixed-length records: Enumerable elements in a file each of which has the same length. For example, in GW-BASIC, the records of a random file have the same length.

floppy disk: Synonymous with diskette.

foreground color: The color of a character (character foreground color), or the color used to draw pictures when no color parameter is specified in a graphics statement (graphics foreground color).

full duplex: A communication system permitting simultaneous operation in both directions.

function: An algorithm returning a single value. A function can be a user or an intrinsic function. It can be called simply by stating its name, followed (in parentheses) by one or more "arguments" representing the values that the function parameters are to assume.

stack: An area of memory in which data items are temporarily stored in a LIFO (Last In First Out) queue.

statement: An instruction to the computer to perform some sequence of operations.

string delimiter: A character that limits a string of characters and therefore cannot be part of the string.

string expression: An expression that returns a string value.

string variable: A simple variable or array element whose value is a string.

subroutine: Either a machine language routine (which may be called by a CALL statement, or an USR function), or a section of a GW-BASIC program (which may be called by a GOSUB or ON... GOSUB statement). At the end of the execution of a subroutine, control is usually returned to the first statement following the calling statement.

subscript: A positive integer which specifies the position of an array element within the array.

text window: A rectangular portion of the screen where text is output. It may be defined by a VIEW PRINT or a WIDTH statement.

tiling: In graphics, a rectangular structure of bits used repeatedly to paint an area with a given pattern of colors.

trap: A special form of a conditional breakpoint that is activated by an event to be intercepted. It also refers to the action to be taken after the interception.

type declaration character: A special character placed at the end of a variable - It may be : % (integer variable), ! (single-precision variable), # (double-precision variable), or \$ (string variable).

type of a variable: Indicates whether the variable is a string or a numeric variable and (if numeric) if it is an integer, a single-precision, or a double-precision variable. The type of a variable may be set by a DEF (INT, SNG, DBL, or STR) statement, or by a character definition tag at the end of the variable name.

GLOSSARY

type of an expression: The type of an expression is the data-type (string, integer, single-precision, or double-precision) of the resulting evaluation of the expression. It depends on the type of its operands.

typewriter keyboard: The central section of the keyboard that is used as a standard typewriter keyboard.

underflow: In an arithmetic operation, the generation of a quantity less than the minimum value different from zero that can be represented in number format. If an underflow occurs GW-BASIC supplies zero, and execution continues.

user function: A function that the user must define before it is called (see DEF FN statement).

variable: A named data item whose values may change during program execution. A variable may be a simple variable or an array element. The name of the variable also specifies its type (numeric or string).

variable-length record: A record whose length is independent of the length of other records in the file.

vector: A collection of variables of the same type under one name that can be ordered in a list. Each element of a vector may be addressed by specifying the value of its subscript.

viewport: A rectangular portion of the screen onto which window contents are mapped. A viewport is defined by a VIEW statement to display both graphics and text. It is possible to have more than one viewport on the screen, but only one will be current.

visual page: The section of the screen buffer which is currently displayed on the screen. It may be different from the active page in Text Mode (see "active page").

wild card character: A special symbol used to represent any single character (?) or any string of characters (*) in a filename.

window: The world coordinate space which is mapped onto the current viewport by the WINDOW statement.

world coordinates: In graphics, the cartesian coordinates of a point of a figure to be displayed, expressed using the user-defined coordinate system.

zooming: The technique of increasing or decreasing the size of the image to be displayed by changing the logical dimensions of the current viewport by the WINDOW statement.

Code 4001320 N. 00
Printed in Italy

NOTICE

Ing. C. Olivetti & C., S.p.A. reserves the right to make any changes in the product described in this manual at any time and without notice.

This manual is licensed to the Customer under the conditions contained in the User License enclosed with the Program to which the manual refers.

Code 4001420 H (5)
Printed in Italy



olivetti

Code 4001420 H (5)
Printed in Italy



olivetti